Storing and Querying Semantic Data in the Cloud

Daniel Janke[1] and Steffen Staab[1,2]

[1] Institute for Web Science and Technologies
Universität Koblenz-Landau, Germany
{danijank,staab}@uni-koblenz.de
http://west.uni-koblenz.de/
[2] Web and Internet Science Group
University of Southampton, UK
s.r.staab@soton.ac.uk
http://wais.ecs.soton.ac.uk/

Abstract. In the last years, huge RDF graphs with trillions of triples were created. To be able to process this huge amount of data, scalable RDF stores are used, in which graph data is distributed over compute and storage nodes for scaling efforts of query processing and memory needs. The main challenges to be investigated for the development of such RDF stores in the cloud are: (i) strategies for data placement over compute and storage nodes, (ii) strategies for distributed query processing, and (iii) strategies for handling failure of compute and storage nodes. In this manuscript, we give an overview of how these challenges are addressed by scalable RDF stores in the cloud.

1 Introduction

In the last years, huge RDF graphs with trillions of triples were created. For instance, the number of Schema.org-based facts that are extracted out of the Web have reached the size of three trillions [98]. Another example is the European Bioinformatics Institute (EMBL-EBI) that would like to convert its datasets into RDF resulting in a graph consisting of trillions of triples. To date no such scalable RDF store exists and the current EBI RDF Platform can handle only 10 billion triples [88].

In order to provide RDF stores that can scale to these huge graph sizes, researchers have started to develop RDF stores in the cloud, where graph data is distributed over compute and storage nodes for scaling efforts of query processing and memory needs. The main challenges to be investigated for such development are: (i) strategies for data placement over compute and storage nodes, (ii) strategies for distributed query processing, and (iii) strategies for handling failure of compute and storage nodes. In this manuscript, we want to give an overview of how these challenges have been addressed by scalable RDF stores in the cloud that have been developed in the last 15 years.

In section 3, we give an overview of the different architectures of scalable RDF stores in the cloud. Basically there exist three types of architectures. The first type uses general cluster computing frameworks like Spark[3] or HBase[4] to perform queries on

[3] https://spark.apache.org/
[4] https://hbase.apache.org/

RDF graphs. The second type – so-called distributed RDF stores — splits the RDF graph into smaller parts that are then stored and queried on the compute and storage nodes. The last type are federated query processing systems. These systems do not have any influence on the data distribution over compute and storage nodes. Instead, they process queries over several RDF stores.

In case of cluster computing frameworks and distributed RDF stores, the RDF graph is distributed among several compute and storage nodes. The general procedure of data distribution strategies is to first split the graph into several not-necessarily disjoint triple sets. In relational and NoSQL databases this splitting is called sharding. Thereafter, the individual triple sets are assigned to compute and storage nodes. An overview of how these two steps are preformed by the existing RDF stores is given in section 4.

Querying a distributed dataset is usually done by splitting the query into subqueries that can be answered locally without the need to exchange data over the network. The results of these subqueries are finally combined into the overall results of the query. In order to identify which parts of the queries can be answered locally on which compute nodes, an index is required that stores information about the data distribution. The different types of indices are described in section 5. An overview of how distributed query processing is done by RDF stores in the cloud is given in section 6.

Another challenge of scalable RDF stores in the cloud is that the failure of an individual compute or storage node should not lead to the failure of the complete RDF store. A brief overview of how this challenge is addressed is given in section 7.

In order to evaluate how well RDF stores in the cloud solve the challenges in the cloud, their performance needs to be evaluated. In section 8, we will present different methodologies how RDF stores in the cloud are evaluated.

Due to the huge amount of RDF stores in the cloud that were developed in the last 15 years, we will not present distributed solutions for the handling of RDF streams or reasoning. Interested readers are referred to [118]. Beside RDF stores in the cloud there also exist distributed graph databases like Sparksee[5] or Titan[6] as well as distributed graph processing frameworks like PGX.D [61] or PEGASUS [66]. They are not described in this manuscript since they have not been presented as part of an RDF store, yet. Furthermore, this manuscript gives an overview of how RDF stores in the cloud work. Readers interested in a performance comparison of RDF stores in the cloud are referred to, e.g., [49].

2 Preliminaries

RDF stores in the cloud have to deal with the two challenges how to distribute RDF graphs on several compute and storage nodes and how to retrieve data thereafter again. The following two sections formalize these challenges. The given definitions are the usual definitions found in the literature. This section was taken from [64].

[5] http://www.sparsity-technologies.com/
[6] http://titan.thinkaurelius.com/

2.1 Formalization of Graph Cover Strategies

In order to illustrate different graph cover strategies, we use Figure 1 as our running example. The graph represents the knows relationship between two employees of the university institute WeST and one employee of the Leibniz institute GESIS. Additionally, the graph includes the ownership of the dog Bello. The terms r:, e:, w:, g:, and f: abbreviate IRI prefixes.

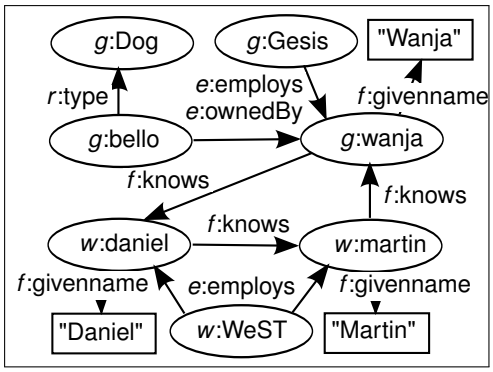


Fig. 1: The example graph describing the knows relationships between some employees of WeST and Gesis.

To formalize the data distribution challenge, we define RDF graphs like in [53]. Assume a signature $\sigma = (I, B, L)$, where I, B and L are the pairwise disjoint infinite sets of IRIs, blank nodes and literals, respectively. The union of these sets is abbreviated as *IBL*.

Definition 1. *The* set of all possible RDF triples T *for signature* σ *is defined by* $T = (I \cup B) \times I \times IBL$. *An* RDF graph G *or simply* graph *is defined as* $G \subseteq T$. *The set of all vertices contained in* G *is defined by* $V_G = \{v | \exists s, p, o : (v, p, o) \in G \vee (s, p, v) \in G\}$.

$(s, p, o) \in T$ is also called a triple with *subject* s, *property* p and *object* o. To simplify later definitions, the functions $\mathrm{subj}(t)$, $\mathrm{obj}(t)$ and $\mathrm{prop}(t)$ return the subject, object or property of triple t, respectively. Likewise, we use $\mathrm{subj}(T)$, $\mathrm{obj}(T)$ and $\mathrm{prop}(T)$ to refer to the set of subjects, objects and properties in the triple set T.

In the context of distributed RDF stores, the triples of a graph have to be assigned to different compute and storage nodes (in the following, we refer to them more briefly as *compute nodes*). The finite set of compute nodes is denoted as C in the rest of this paper.

Definition 2. *Let* G *denote an RDF graph. Then a* graph cover *is a function* $\mathrm{cover} : G \rightarrow 2^C$, *that assigns each triple of a graph* G *to at least one compute node.*

Definition 3. *The function* chunk *returns the triples assigned to a specific compute node by a graph cover (*graph chunks*). It is defined as*

$$\text{chunk}_{\text{cover}}\colon C \rightarrow 2^G$$
$$\text{chunk}_{\text{cover}}(c) := \{t | c \in \text{cover}(t)\} \quad .$$

This definition allows for triples being replicated on several compute nodes. If the graph chunks are pairwise disjoint the underlying graph cover is called a *graph partitioning*.[7]
Some frequently used graph cover strategies require two additional definitions.

Definition 4. *A graph cover of RDF graph G is* subject-complete*, if* $\forall c \in C : \forall (s,p,o) \in \text{chunk}_{\text{cover}}(c) : \forall (s,p',o') \in G : (s,p',o') \in \text{chunk}_{\text{cover}}(c)$.

Example 1. The graph cover shown in Figure 2 is subject-complete, since all triples with the same subject are located in the graph chunk of c_1 or c_2.

Definition 5. *A* path P *is a sequence* $\langle t_0, t_1, ..., t_n \rangle$*, if* $\forall i \in [0,n] : t_i \in G \wedge \forall j \in [0,n] : j \neq i \Rightarrow t_j \neq t_i$ *and* $\forall i \in [1,n] : t_{i-1} = (s_{i-1}, p_{i-1}, s_i) \wedge t_i = (s_i, p_i, o_i)$. *The* length of path P *is* $n+1$.

Example 2. In the example Graph shown in Figure 1, $\langle$(w:daniel, f:knows, w:martin), (w:martin, f:knows, g:wanja)$\rangle$ is a path of length 2.

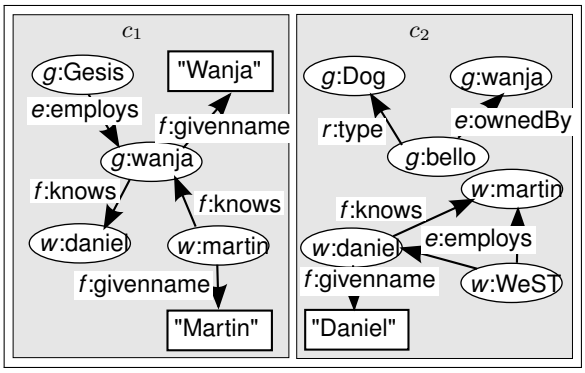


Fig. 2: An example graph cover of the example graph.

Several RDF stores in the cloud create subsets of an RDF graph that are distributed among the compute nodes. One frequently used graph subset is called a molecule. A molecule is a set of triples that are reachable from its anchor vertex. This property of

[7] In the context of relational or NoSQL databases, graph covers are called sharding and the graph chunks shards. In the literature, there exist definitions of sharding that allow for data replication whereas others do not allow it.

molecules aims for an efficient processing of star-shaped queries, in which the triple patterns are joined on the subject. Also path-shaped queries might be processed efficiently, if the length of the path is smaller than the diameter of the molecule.

Definition 6. *A subset $M_v \subseteq G$ of an RDF graph G is called* molecule[8] of vertex v *if*

1. *for all triples $t \in M_v$ there must exist a path $\langle t_0, t_1, ..., t_n, t \rangle$ such that $t_0 = (v, p_0, o_0)$ and $t_0, t_1, ..., t_n \in M_v$ and*
2. *if s is a subject of some triple in M_v, then $\forall (s, p, o) \in G : (s, p, o) \in M_v$.*

The vertex v is called anchor vertex[9].

Definition 7. *The* directed molecule diameter *of a molecule M_v is the longest shortest path between the anchor vertex v and all objects contained in triples of M_v.*

Example 3. The molecule with directed diameter 1 of the anchor vertex g:gesis in the example graph only contains the triple (g:Gesis, e:employs, g:wanja). A molecule with diameter 2 of the same anchor vertex would additionally contain (g:wanja, f:knows, w:daniel) and (g:wanja, f:givenname, "Wanja").

2.2 Formalization of Query Execution Strategies

We define the used SPARQL core as done in [110], [107] and [16]. For this definition the infinite set of variables V that is disjoint from *IBL* is required. In order to distinguish the syntax of variables from other RDF terms, they are prefixed with ?. The syntax of SPARQL is defined as follows.

Definition 8. *A* triple pattern *is a member of the set TP $= (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$.*

Definition 9. *A* basic graph pattern *(BGP) is a*

1. *triple pattern or*
2. *a conjunction $B_1.B_2$ of two BGPs B_1 and B_2.*

Definition 10. *A* SELECT query *is defined as* `SELECT` W `WHERE` $\{B\}$ *with $W \subseteq V$ and B a BGP.*

Example 4. The following SELECT query returns the names of all persons who are known by employees of WeST and who own the dog Bello. It contains a basic graph pattern that concatenates four triple patterns. In the following examples `?v1 <f:knows> ?v2` is abbreviated as tp_1, `?v2 <f:givenname> ?v3` as tp_2 and so on. All following examples in this section will refer to this query.

```
SELECT ?v3 WHERE {
    ?v1 <f:knows>  ?v2.
    ?v2 <f:givenname>  ?v3.
    <w:WeST> <e:employs> ?v1.
    <gs:bello> <e:ownedBy> ?v2
}
```

[8] We adapted the definition of an RDF molecule in [38] to allow for paths with a length ≥ 1.

[9] The term anchor vertex was taken from [79].

Before the semantics of a SPARQL query can be defined, some additional definitions are required. In the following $\mathcal{Q}$ represents the set of all SPARQL queries.

Definition 11. *The function* $\mathrm{var} : \mathcal{Q} \rightarrow V$ *returns the set of variables occurring in a SPARQL query. It is defined as:*

1. $\mathrm{var}(tp)$ *is the set of variables occurring in triple pattern tp.*
2. $\mathrm{var}(B_1.B_2) := \mathrm{var}(B_1) \cup \mathrm{var}(B_2)$ *for the conjunction of the BGPs* B_1 *and* B_2.
3. $\mathrm{var}(\texttt{SELECT}\ W\ \texttt{WHERE}\ \{B\}) := W \cap \mathrm{var}(B)$ *for* $W \subseteq V$ *and* B *a BGP.*

Definition 12. *A* variable binding *is a partial function* $\mu : V \nrightarrow IBL$. *The* set of all variable bindings *is* $\mathcal{O}$.

The abbreviated notation $\mu(t)$ with $t \in TP$ means that the variables in t are substituted according to μ.

Example 5. The following three partial functions are variable bindings, that assign values to some variables. μ_1 would be an intermediate result produced by the first triple pattern of the example query in example 4 whereas μ_2 and μ_3 would be produced by the second triple pattern.

$$\mu_1 = \{(?v_1, \mathsf{w{:}martin}), (?v_2, \mathsf{g{:}wanja})\}$$
$$\mu_2 = \{(?v_2, \mathsf{g{:}wanja}), (?v_3, \text{"Wanja"})\}$$
$$\mu_3 = \{(?v_2, \mathsf{w{:}martin}), (?v_3, \text{"Martin"})\}$$

Definition 13. *Two variable bindings* μ_i *and* μ_j *are* compatible*, denoted by* $\mu_i \sim \mu_j$*, if* $\forall ?x \in \mathrm{dom}(\mu_i) \cap \mathrm{dom}(\mu_j) : \mu_i(?x) = \mu_j(?x)$.[10]

Example 6. The variable bindings μ_1 and μ_2 from example 5 are compatible since in both variable bindings g:wanja is assigned to $?v_2$ which is the only variable occurring in the domains of both variable bindings. μ_1 and μ_3 as well as μ_2 and μ_3 are not compatible because they assign different values to common variables.

Definition 14. *The* join *of two sets of variable bindings* Ω_1 *and* Ω_2 *is defined as* $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \sim \mu_2\}$.
The variables contained in $\mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$ *are called* join variables.

Example 7. The join of the two variable bindings sets $\{\mu_1\}$ and $\{\mu_2, \mu_3\}$ from example 5 produces a result set only containing the variable binding $\{(?v_1, \mathsf{w{:}martin}), (?v_2, \mathsf{g{:}wanja}), (?v_3, \text{"Wanja"})\}$ because only μ_1 and μ_2 are compatible.

[107] and [16] define the semantics of a SPARQL query as follows:

Definition 15. *The* evaluation *of a SPARQL query* Q *over an RDF Graph* G*, denoted by* $[\![Q]\!]_G$*, is defined recursively as follows:*

1. *If* $tp \in TP$ *then* $[\![tp]\!]_G = \{\mu | \mathrm{dom}(\mu) = \mathrm{var}(tp) \wedge \mu(tp) \in G\}$.
2. *If* B_1 *and* B_2 *are BGPs, then* $[\![B_1.B_2]\!]_G = [\![B_1]\!]_G \bowtie [\![B_2]\!]_G$.

[10] $\mathrm{dom}(\mu)$ refers to the set of variables of this binding.

3. *If* $W \subseteq V$ *and* B *is a BGP, then* $[\![\texttt{SELECT}\ W\ \texttt{WHERE}\ \{B\}]\!]_G = \text{project}(W, [\![B]\!]_G) = \{\mu_{|_W} | \mu \in [\![B]\!]_G\}$.[11]

The execution of a query requires the translation of a SPARQL query into a query execution tree. This tree defines the individual operations and their execution sequence. Thereby, each node of the query execution tree consists of three components: (i) the name of the operation to be executed, (ii) the set of variables that are bound in the resulting variable bindings and (iii) the set of child operations.

Definition 16. *Let* L_{node} *be the set of node labels and* $\Upsilon = L_{node} \times 2^V \times 2^\Upsilon$ *the set of all query execution trees, then a* query execution tree *of a query* Q*, denoted as* $\langle\!\langle Q \rangle\!\rangle$*, is defined recursively as follows:*

1. *If* $tp \in TP$ *then* $\langle\!\langle tp \rangle\!\rangle = (tp, var(tp), \varnothing)$.
2. *If* B_1 *and* B_2 *are BGPs, then* $\langle\!\langle B_1.B_2 \rangle\!\rangle = (join, var(B_1) \cup var(B_2), \{\langle\!\langle B_1 \rangle\!\rangle, \langle\!\langle B_2 \rangle\!\rangle\})$.
3. *If* $W \subseteq V$ *and* B *is a BGP, then* $\langle\!\langle \texttt{SELECT}\ W\ \texttt{WHERE}\ \{B\} \rangle\!\rangle = (project, W, \langle\!\langle B \rangle\!\rangle)$.

Example 8. Figure 3 shows a graphical representation of one query execution tree for the example query from example 4. The following query execution tree represents the first join in its mathematical representation. It has the two child trees $(tp_1, \{?v_1, ?v_2\}, \varnothing)$ and $(tp_2, \{?v_2, ?v_3\}, \varnothing)$.

$$
\begin{aligned}
&(\text{join}, \{?v_1, ?v_2, ?v_3\}, \{ \\
&\qquad (tp_1, \{?v_1, ?v_2\}, \varnothing), \\
&\qquad (tp_2, \{?v_2, ?v_3\}, \varnothing) \\
&\})
\end{aligned}
$$

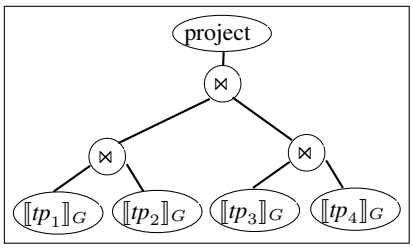


Fig. 3: Bushy query execution tree for the query from example 4.

3 Architectures

The RDF stores in the cloud can be categorized into three groups that characterize their architecture. The first type of RDF stores in the cloud make use of cloud computing

[11] $\mu_{|_W}$ means that the domain of μ is restricted to the variables in W.

frameworks (see Sec. 4). These frameworks hide the complexity of distributed systems from the developers. This reduced developing complexity comes with the cost of limited influence on, e.g., the data placement. To overcome these limitation, developers of distributed RDF stores have to address the challenges of data placement, distributed query processing and fault tolerance on their own (see Sec. 3.2). In contrast to this, federated RDF stores aim to query data from several RDF stores that manage the stored data on their own (see Sec. 3.3). One application scenario would be querying several remote SPARQL endpoints that can be found in the linked open data cloud.

One RDF store in the cloud that caused a lot of attention after its launch is Neptune[12]. Due to a lack of descriptions that can be found, it is unclear which architecture it has.

3.1 RDF Stores Using Cloud Computing Frameworks

Implementing a distributed system is a challenging task. To reduce the complexity, several RDF stores in the cloud are realized on top of cloud computing frameworks. As shown in Fig. 4, these RDF stores need a master node that translates the RDF graph into some format that can be stored within the cloud computing frameworks. This is the job of the graph converter. Similarly, SPARQL queries need to be translated into queries or tasks that can be executed on the cloud computing framework to produce the query results that are then transferred back to the user. This is done by the query translator.

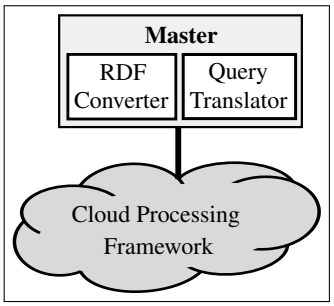


Fig. 4: Architecture of an RDF store using a cloud computing framework.

One of the first cloud computing frameworks that was used to build RDF stores in the cloud is Hadoop[13] [135]. RDF stores like SHARD [116], HadoopRDF [42] and CliqueSquare [44] transform the RDF graph into one or several files that are stored in the distributed file system of Hadoop [128]. SPARQL queries are translated into one or several jobs that are executed within Hadoop. Depending on how the RDF graph is separated into files, one or several files are processed during query execution.

[12] `https://aws.amazon.com/neptune/`
[13] `https://hadoop.apache.org/`

To simplify the usage of Hadoop, the high-level query language and execution framework Pig[14] [99] was developed on top of Hadoop. Instead of translating SPARQL into Hadoop jobs directly, systems like PigSPARQL [122] and RAPID+ [72] translate SPARQL into the Pig query language. Pig then translate the code into Hadoop jobs and tries to optimized the orchestration of the individual jobs.

One of the main limitations of Hadoop is that the result of each individual task has to be written back into the distributed file system. To overcome this limitation, Spark[15] [144] was developed. Spark stores the result of each job in main memory. These results can be used by several other jobs before the final result is optionally persisted on disk. Spark is used by SPARQLGX [50], S2RDF [124], SPARQL-Spark [96] and PRoST [29].

On top of Spark the graph processing framework GraphX[16] [45] was developed. In this framework a graph can be loaded and algorithms can be performed on it. These algorithms are vertex-centric, i.e., each vertex is able to receive, process and send messages to its neighboured vertices. S2X [121] translates SPARQL queries into such vertex-centric algorithms to produce the query results. Similar to S2X TripleRush [130], Random Walk TripleRush [132] (both basing on Signal/Collect [131]) and [46] use vertex-centric graph processing frameworks.

Another type of cloud computing frameworks are NoSQL (Not only SQL) database systems. These systems usually scale well with a high number of compute nodes and a fault tolerant. Their withdraw is that they usually do not provide ACID transactions. One type of NoSQL systems are key-value stores. These systems map a unique key to an arbitrary value. DynamoDB[17] [34] is such a distributed key-value store. It is used in AMADA [24] to index and store the triples of the RDF graph.

Column-family stores are another type of NoSQL database systems. These systems store tabular data like in relational databases. Instead of storing all data of a row physically together, column-family stores locate all entries of a set of columns (i.e., a column-family) physically together. Examples of these stores are HBase[18] (used by Jena-HBase [70], H_2RDF+ [103]), Cassandra[19] [77] (used by CumulusRDF [75]), Accumulo[20] (used by Rya [114] and RDF-4X [4]) and Impala[21] [117] (used by Sempala [123] and [111]). RDF stores relying on these column-family stores usually vary in the way how they store the RDF graph in tables.

The last type of NoSQL database systems that are used in RDf stores are document stores. Document stores store the data in documents which are objects with arbitrary fields and values. Each of the fields can be used to index the data. The RDF store [32] uses Couchbase[22] and D-SPARQ [95] uses MongoDB[23].

[14] https://pig.apache.org/
[15] https://spark.apache.org/
[16] https://spark.apache.org/graphx/
[17] https://aws.amazon.com/de/dynamodb/
[18] https://hbase.apache.org/
[19] https://cassandra.apache.org/
[20] https://accumulo.apache.org/
[21] https://impala.apache.org/
[22] https://www.couchbase.com/
[23] https://www.mongodb.com/

3.2 Distributed RDF Stores

In contrast to RDF stores that use cloud computing frameworks, distributed RDF store have to address the challenges resulting from the distribution. To reduce the complexity of these challenges, most distributed RDF stores are realized with a master-slave architecture. In this architecture a dedicated compute node is the master. It is responsible for coordinating the individual slaves that store the RDF graph and process the queries. The disadvantage of this architecture is that the master node can easily become a bottleneck of the distributed RDF store. To overcome this limitation some distributed RDF stores are realized with a peer-to-peer architecture in which the design of every compute node is identical.

Distributed RDF Stores with Master-Slave Architecture. In distributed RDF stores that have a master-slave architecture there exists one master and several slaves. The master is a dedicate compute node that is responsible for the coordination of all slaves. The slaves are the compute nodes that store the graph and process the queries. The general architecture of such a distributed RDF store is shown in Figure 5.

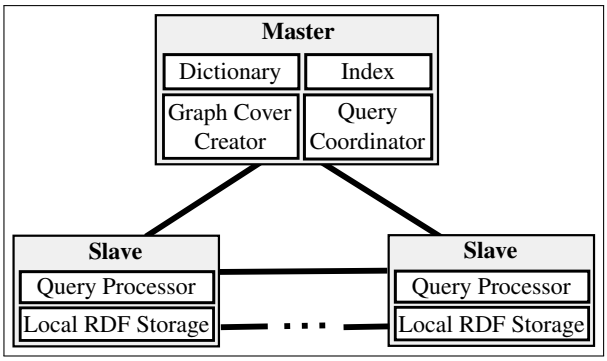


Fig. 5: Architecture of a master-slave distributed RDF store.

When a graph is loaded, the master first replaces each string identifier of a resource by a shorter unique integer identifier. This replacement reduces the storage size of the graph that needs to be processed. The mapping between the string and integer identifiers are stored in the dictionary. Thereafter, the graph cover creator assigns the triples to the individual slaves. Thereby, an index is created which keeps track on which slave which part of the graph is stored. Each slaves stores the triples assigned to him in its local RDF storage.

In order to query the RDF store, a query is send to the master. The query coordinator first encodes all string identifier in the query with the help of the dictionary. The query is then translated into a query execution tree and optimized. With the help of the index, the query coordinator can decide which part of the query can be executed on which slave and initiates the query processing on the slaves. On each slave the query processor

processes the (sub)query assigned to him on the local RDF storage. The intermediate results can then be directly exchanged between all slaves. The final results are sent back to the query coordinator. With the help of the dictionary the query coordinator replaces the integer identifier of the results by it string identifier and sends them back to the user.

The architectures of Custered TDB [102], COSI [23], Partout [43], GraphDB [17], Blazegraph [2], SemStore [138], TriAD [52], DREAM [55], [105], DiploCoud [139] and [104] are as described above. But there also exist variations of this architecture. For instance in Trinity.RDF [145], WARP [62], YARS2 [60] and 4store [58] the query coordinator also has to join intermediate results.

Some of the components of the master can be distributed among the slaves. For instance, in EAGRE [146] the graph cover creation is performed on all slaves in parallel and the index is distributed over all slaves. Furthermore, distributed RDF stores do not necessarily need all of the presented components. For instance, PHDStore [10], 2way [106] and [27] do not need a global index and/or dictionary. If distributed RDF stores adapt the graph cover during runtime based on the actual workload, the master also contains a redistribution controller as in AdPart [57], AdHash [9, 56] and Spartex [5].

Beside these pure distributed RDF stores there also exist hybrid RDF stores that also use some cloud computing infrastructure. In Sedge [142] a reimplementation of Pregel [84] is used to process distributed queries. A more common approach is, to use Hadoop to process only distributed joins as done in [63], [39], [143], SPA [81], VB-Partitioner [79], SHAPE [80] and [137].

Distributed RDF Stores with peer-to-peer Architecture. In distributed RDF stores that follow the peer-to-peer architecture all compute nodes – called peers – consist of the same components. This architecture has the advantage that no single compute node can become a bottleneck by design. The disadvantage is that usually no compute node knows how many compute nodes exist and how the data is distributed among all compute nodes. Usually in systems like Edutella [37, 97], RDFPeers [25], PAGE [35], GridVine [6, 31], RDFCube [87], Atlas [67], 3RDF [12], [13] and [101], a distributed index is used that routes requests to the compute node storing the requested data. Since this distributed index is the central component of the architecture, the implementation choice of the index determines how the triples of the RDF graph are assigned to the compute nodes. The architecture of most peer-to-peer distributed RDF stores is shown in Figure 6.

To load an RDF graph, the complete graph can be sent to any compute node. The graph cover creator asks the distributed index on which compute node the individual triples should be created. Based on this decision the triples are distributed over the compute nodes. When a compute node receives triples from a graph cover creator, it inserts them into its local RDF store and updates the distributed index. In order to speed up the graph loading procedure, the RDF graph can be split into several parts that are processed by the graph cover creators on several compute nodes in parallel.

When a user sends a query to any of the compute nodes, the query coordinator creates the query execution tree. Based on the index the query coordinator decides which part of the query execution tree is sent to which compute node. When a query processor

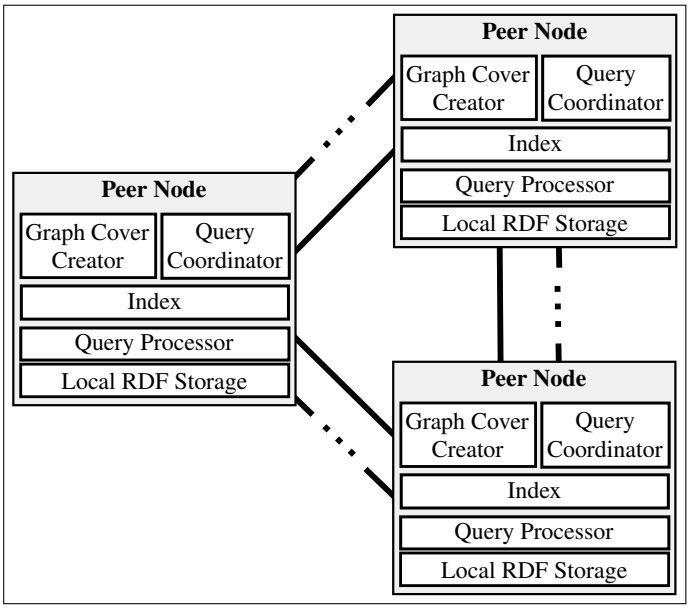


Fig. 6: Architecture of a peer-to-peer distributed RDF store.

receives some subquery from a query coordinator, it retrieves the requested information from its local RDF store. The intermediate results are sent to other compute nodes or back to the query coordinator where they will be further processed. Finally, the query coordinator computes the overall results and send them back to the user.

In contrast to the master-slave architecture, peer-to-peer distributed RDF stores usually do not have a global dictionary. As a consequence string identifiers are used when transferring triples during the loading of a graph or intermediate results during the query processing.

3.3 Federated RDF Stores

In the linked open data cloud[24] there exist many public datasets and SPARQL endpoints. In order to combine several datasets and query them together a naive approach would be to download all datasets and load them into a single RDF store. This naive approach has several disadvantages. First of all, it requires a lot of computational resources to store and process several datasets at once. Furthermore, when the datasets change, the system would need to keep track of these changes and update its local copies of the datasets.

To overcome these limitations, federated RDF stores have been developed. Their basic idea is to query remote RDF stores directly. The general architecture of federated RDF stores is shown in Figure 7. In order to query the remote RDF stores a global index

[24] `http://lod-cloud.net/`

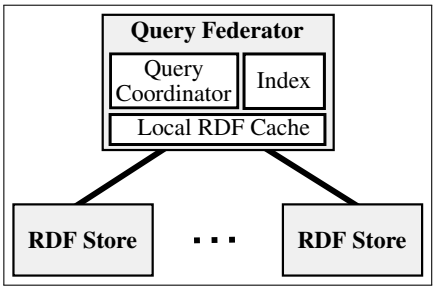


Fig. 7: Architecture of a federated RDF store.

is required that indicates which RDF stores contain data that are relevant for the query. This index is created by retrieving statistical information that are provided by the remote RDF stores (e.g., SPLENDID [48], WoDQA [8], LHD [134], DAW [120], SemaGrow [26], FEDRA [91], LILAC [92] and Odyssey [90]) or the user (e.g., DARQ [115]), by sending special queries to the remote RDF stores (e.g., FedX [127], ANAPSID [7, 93], Lusail [86]) or by observing the results that are returned during the processing of user queries (e.g., ADERIS [83]). Also combinations of these strategies are possible as in Avalanche [18]. Also the absence of the index is possible if the resource IRIs are dereferenced as proposed by SIHJoin [76].

When a user sends a query, the query coordinator transforms it into a query execution tree. With the help of the index it can decide from which remote RDF store can contributed to the query. Based on this decision it forwards the query or parts of it to the corresponding remote RDF stores. The returned intermediate results are joined by the query coordinator and finally sent back to the user. In order to speed up the query execution, the query federator also contains a local RDF cache in which data retrieved by previous queries is cached. This cached data can be reused for future queries, which might reduce the number of queries that needs to be sent to the remote RDF stores.

4 Graph Cover Strategies

One core aspect of RDF stores in the cloud is, how the RDF graph is distributed among the compute nodes, resulting in a graph cover. A common procedure to create a graph cover is to first split the RDF graph into small possibly overlapping subsets. Thereafter, this graph subsets are assigned to compute nodes. In RDF stores that bases on cloud computing frameworks, the influence on how this assignment to compute nodes is done is usually limited. Therefore, in section 4.1 is described how a graph is split into subsets that are then stored in the cloud computing framework. The graph cover strategies of distributed RDF stores can in general be separated into three categories: (i) hash-based graph cover strategies (see section 4.2), (ii) graph-clustering-based graph cover strategies (see section 4.3) and (iii) workload-aware graph cover strategies that distributes the graph based on a historic query workload (see section 4.4). In order to reduce the amount of queries that need to combine data from different compute nodes, the n-hop

replication was proposed that replicates triples at the border of the chunks of arbitrary graph cover strategies (see section 4.5).

When RDF stores are queried frequently, the initial distribution of the graph on the compute nodes might be suboptimal for the current query workload. To improve the query performance, some RDF stores have implemented a dynamic graph cover strategy (see section 4.6). This strategy observes the current query workload and tries to optimize the data placement by moving or copying parts small triple sets from one compute node to another.

Since there are is huge number of graph cover strategies, we focus in this section only on the most frequently graph cover strategies and give only hints to a few variations that can be found. We do not present exotic graph cover strategies that were only used in a single RDF store. Parts of this section was taken from [64].

4.1 Graph Cover Strategies in Cloud-Computing-Framework-Based RDF Stores

RDF stores that build on cloud computing frameworks have usually limited influence how the data is placed on the individual RDF stores. There influence is limited to the way how the RDF graph is split into subsets that are stored in files or tables. The goal of splitting the RDF graph into subsets is to reduce the amount of triples that need to be processed during the query execution. This is achieved by storing all triples with the same subject, object, predicate or combinations of them in the same file or table.

Molecule Graph Splits. In order to process star-shaped queries efficiently, the RDF graph is split into molecules of diameter 1. This means that all triples with the same subject are stored in one file. D-SPARQ [95], follows this approach. The advantage of the molecule graph cover is that star shaped queries whose triple patterns are joined on a subject no join needs to be processed. In case of a constant subject only a single file needs to be processed.

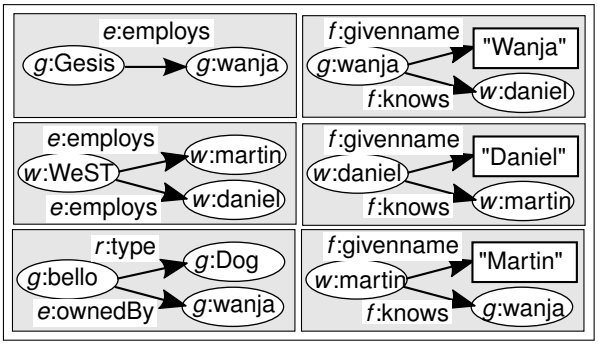


Fig. 8: The example graph split into molecules.

In order to reduce the number of required joins, RAPID+ [72] proposes to store all triples that have the same resource identifier at a subject or object position in a single file. Additionally, RAPID+ reduces the number of joins by increasing the molecule diameter in order to process path-shaped queries more efficiently.

In some RDF stores like Stratustore [129], Sempala [123] and SHARD [116] all molecules are stored in a single table. Each molecule is basically represented by a single row in this table with the subject as unique identifier. The predicates are the column names and the object as the value stored in each cell. If the combination of subject and property occurs in several triples, these cells store a list of objects or several rows for this subject are created. This storage layout is called property table in the literature.

Vertical Graph Splits. The basic idea of the vertical cover originated in [3] to store RDF data in a relational database so that for each property a table is created in which all triples with this property are stored. In the context of distributed RDF stores, approaches like Jena-HBase [71], PigSPARQL [122], [147] and SPARQLGX [124] store all triples with the same property in a file or table. The advantage is that it is easy to compute but a query that matches with paths of length l will only match with triples on at most l compute nodes. Thus, this graph cover strategy is likely to result in an imbalanced workload and a high number of exchanged intermediate results.

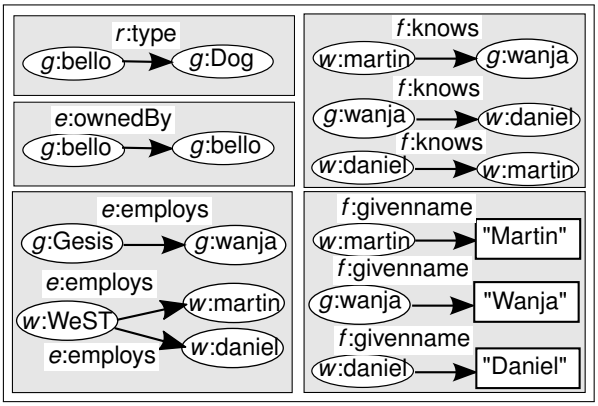


Fig. 9: An example vertical graph split of the example graph.

One disadvantage of the vertical graph split as presented above is, that frequently occurring properties like rdf:type lead to very large files. Therefore, the RDF store HadoopRDF [42] splits these tables based on combinations of properties and the RDFS types of the objects.

Another variant of the vertical graph split is realized in S2RDF [124]. In order to reduce the number of joins that need to be processed, they additionally create tables for all possible subject-subject and subject-object joins of triples.

4.2 Hash-Based Graph Cover Strategies

Hash Cover. A hash cover assigns triples to chunks according to the hash value computed on their subjects modulo the number of compute nodes. Thus, all triples with the same subject – i.e., a molecule – are located in the same graph chunk. This graph cover strategy is used, for instance, by Virtuoso Clustered Edition [40], VB-Partitioner [79], SPA [81], 4store [58] and AdHash [9].

Example 9. The following hash function produces the graph cover shown in Figure 2.

$$\forall r \in \{\mathsf{g{:}Gesis}, \mathsf{g{:}wanja}, \mathsf{w{:}martin}\}\colon \mathrm{hash}(r) := 1$$
$$\forall r \in \{\mathsf{g{:}bello}, \mathsf{w{:}WeST}, \mathsf{w{:}daniel}\}\colon \mathrm{hash}(r) := 2 \ .$$

The advantages of the hash cover are that it is easy to compute and due to a relatively random assignment of triples to compute nodes the resulting graph chunks will have similar sizes. The disadvantages are that it may lead to a high number of exchanged intermediate results if a query matches with long paths. Since all hash covers are subject-contained, this graph cover strategy might be a good choice if the expected queries will only match with paths of a short length (ideally 1).

Beside the subject, distributed RDF stores like Trinity.RDF [145], Clustered TDB [102], YARS2 [59, 60] and RDFPeers [25] also use property and/or the object to assign each triple three times to the compute nodes.[25] Additionally, RDF stores like PAGE [35] and [13] append at least two elements of each triple and use the hash of the result to assign triples to compute nodes.

Hierarchical Hash Cover. Inspired by the observations that IRIs have a path hierarchy and IRIs with a common hierarchy prefix are often queried together, SHAPE [80] uses an improved hashing strategy to reduce the inter-chunk queries. First, it extracts the path hierarchies of all IRIs. For instance, the extracted path hierarchy of "`http://www.w3.org/1999/02/22-rdf-syntax-ns#type`" is "`org/w3/www/1999/02/22-rdf-syntax-ns/type`". Then, for each level in the path hierarchy (e. g., "`org`", "`org/w3`", "`org/w3/www`", ...) it computes the percentage of triples sharing a hierarchy prefix. If the percentage exceeds an empirically defined threshold and the number of prefixes is equal or greater to the number of compute nodes at any hierarchy level, then these prefixes are used for the hash cover.

Example 10. Assume the hash is computed on the prefixes gesis and west of the subject IRIs in the example graph. If the hash function returns 1 for gesis and 2 for west the resulting hierarchical hash cover is shown in Figure 10.

In comparison to the hash cover the creation of a hierarchical hash cover requires a higher computational effort to determine the IRI prefixes on which the hash is computed. For queries that match with paths in which the subjects and objects have the same IRI prefix the number of exchanged intermediate results may be reduced. This

[25] If the hash cover is only computed on the predicate, the resulting graph cover would be similar to the vertical graph split.

reduction might come at the cost of a more imbalanced query workload since only a few chunks will contain these paths. Thus, the use of the hierarchical hash cover might be beneficial (i) if the network connecting the compute nodes is slow or (ii) if other functionality such as prefix matching benefits from the hierarchical hash cover.

4.3 Graph-Clustering-Based Graph Cover Strategies

Graph clustering considers splitting a graph into partitions, i.e., a graph cover with pairwise disjoint graph chunks. In this area a wide variety of algorithms were developed (for instance, see the survey [85]). The basic idea is that an RDF graph is partitioned by one of these algorithms. Since most of the graph clustering algorithm create an assignment from vertices to compute nodes, the triple are usually assigned to the compute node to which its subject was assigned to. The most frequently applied graph clustering approach is the minimal edge-cut partitioning which described below.

Another rarely used graph clustering algorithm tries to optimize the modularity. The modularity measures the difference between the actual number of edges within the partitions and the expected number of such edges. This strategy was applied for instance by MO+ [113].

Minimal Edge-Cut Cover. The minimal edge-cut cover is a vertex-centred partitioning which tries to solve the k-way graph partitioning problem as described in [69]. It aims at minimizing the number of edges between vertices of different partitions under the condition that each partition contains approximately $\frac{|V_G|}{k}$ many vertices. Details about the computation of k-way graph partitioning and the targeted approximation can, e.g., be found in [69]. RDF stores like D-SPARQ [95], [63] and [105] convert the outcome of the minimal edge-cut algorithm, i.e., a partitioning of V_G, into a graph cover of G by assigning each triple to the compute node to which its subject has been assigned.

Example 11. A minimal edge-cut algorithm might assign the resources g:Dog, g:Gesis, g:bello, g:wanja and "Wanja" to compute node c_1 and all other resources to compute

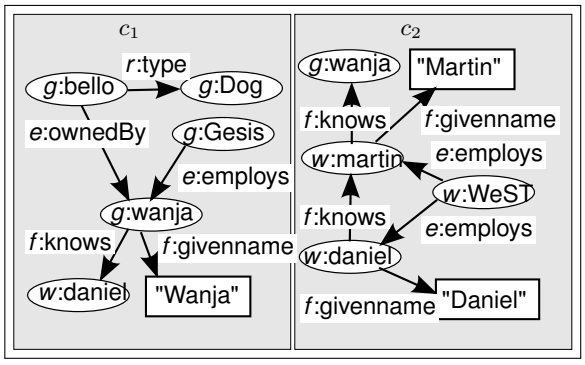


Fig. 10: An example hierarchical hash cover which is also a minimal edge-cut cover of the example graph.

node c_2. For our specific running example the result of the minimal edge-cut cover strategy is identical to the results of the hierarchical hash cover strategy depicted in Figure 10.

In this example there exist two edges connecting vertices assigned to different chunks. One is the f:knows edge starting at g:wanja and ending at w:daniel. The other is the f:knows edge starting at w:martin and ending at g:wanja. Since the subject g:wanja of the first triple is assigned to c_1, this triple is assigned to c_1. The subject of the second triple w:martin is assigned to c_2. Therefore, this triple is assigned to c_2.

Since the minimal edge-cut cover considers the graph structure, the creation of the graph cover requires a high computational effort. The advantage of considering the graph structure might be a reduced number of exchanged intermediate results. This would make the minimal edge-cut cover a good choice if the network connection between compute nodes is slow.

In order to optimize the query performance, TriAD [52] creates an over-partitioning. For instance, to create a graph cover that assigns triples to 5 compute nodes, 100k-200k partitions are created. These partitions are then assigned to compute nodes. To improve the performance of queries that use RDFS schema information, in [108] the RDFS schema is replicated to all graph chunks. Since the minimal edge-cut can lead to graph chunks whose cardinally vary strongly, [133] proposes to weight vertices by the number of triples in which it occurs.

An alternative optimization is performed by EAGRE [146]. Instead of partitioning the original graph, a minimal edge-cut cover of the summary graph is created. Each vertex of the summary graph represents a set of molecules that have similar predicates. An edge (v_1, v_2) in the summary graph is created, if any anchor vertex of the molecules contained in v_2 occur as object in any molecule contained in v_1. The vertices are weighted by the number of molecules they contain. This graph cover strategy ensures that molecules with similar predicates are stored on the same compute node.

4.4 Workload-Aware Graph Cover Strategies

Another type of graph cover strategies assume that the query workload does not change much over the time. Therefore, they learn from a historic query workload which triples have been frequently queried together first. Based on this knowledge they try to find a optimal graph cover for future queries. These approaches are, for instance:

- The novel idea applied in WARP [62] is creating an initial minimal edge-cut cover and then replicate triples in a way such that all historic queries can be answered locally.
- In COSI [23] edges are weighted based on the frequency they are requested by the historic query workload. Thereafter, a weighted minimal edge-cut partitioning is performed leading to an improved horizontal containment.
- In [17] the resulting graph cover aims to balance the overall workload of all queries equally among all compute nodes. Thereby, each query is processed by a single compute node in an ideal case. To reach this goal, the proposed algorithm assigns the triples required by the queries to compute nodes in a way that the number of replicated triples is reduced.

– In Partout [43] the queries contained in the historic query workload are first generalized by replacing every rarely queried subject or object constants by variables. Thereafter, the matches of this generalized triple patterns are assigned to compute nodes in a way that (i) ideally each query can be answered by a single compute node without replicating triples and (ii) the query workload of all queries is distributed equally among all compute nodes.
– DiploCloud [140] generalizes the queries in the historic query workload by using schema information. Then triple sets are computed that can produce a single query result. Finally, the triple sets are distributed equally among all compute nodes.

4.5 n-Hop Replication

Whenever a query combines data from different graph chunks, intermediate results need to be exchanged between different compute nodes. To reduce the number of exchanged intermediate results for a subject-complete graph cover of graph G, the n-hop replication strategy extends each of its chunks ch_i by replicating all triples contained in some path of length $\leq n$ in G starting at some subject or object occurring in ch_i. This way all queries that match with paths of length $\leq n$ could be processed without exchanging intermediate results. The n-hop replication is used by systems like [63], VB-Partitioner [79] and D-SPARQ [95].

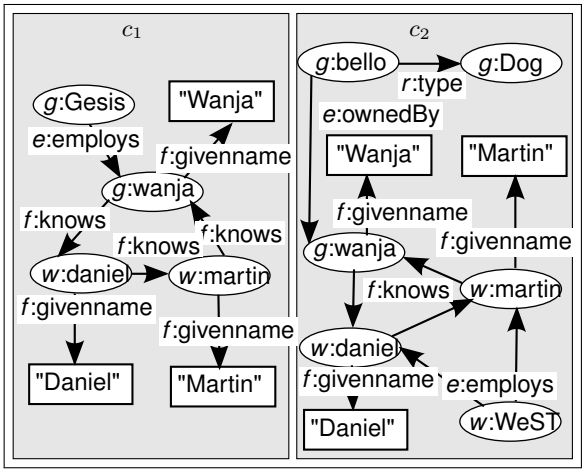


Fig. 11: The 2-hop extension of the hash cover in Figure 2.

Example 12. Applying the 2-hop replication extension on the hash cover in Figure 2 results in the 2-hop hash cover shown in Figure 11. In this cover a query could match with the path $(\text{g:bello}, \text{e:ownedBy}, \text{g:wanja}), (\text{g:wanja}, \text{f:knows}, \text{w:daniel})$ on compute node c_2 without the need to exchange intermediate results.

The n-hop replication may reduce the number of transferred intermediate results at the cost of replicating triples. This replication will increase the effort to create the graph cover and increase the size of the graph chunks. Furthermore, the replication might cause a higher computational effort during the query processing since the replicated triples might lead to duplicate intermediate results. Thus, using the n-hop replication might be beneficial if the network connecting the intermediate results is slow and the number of replicated triples is low.

4.6 Dynamic Graph Cover Strategies

Graph covers created by one of the graph cover strategies above can lead to a high amount of data transfer between compute nodes, if the actual query workload needs to combine data stored on different compute nodes. In order to overcome this limitation, PHD-Store [9] and AdHash [9] keeps track of basic graph patterns that are queried frequently. When the frequency exceeds a threshold, triples that match with these frequent triple patterns are replicated in a way that these basic graph patterns can be executed locally.

Instead of only trying to reduce the network communication, Sedge [9] tries to primarily distribute the query workload equally among the compute nodes. Therefore, Sedge keeps track how frequently the molecules are queried together. If a set of molecules are queried together with a high frequency, these set of molecules is replicated to a compute node with a low workload.

Another type of graph cover strategies assumes that during runtime new triples can be added to the RDF store. In this setting it may happen that a single compute node stores much more triples then other compute nodes. To prevent the compute node from being overloaded, the triples of that compute node can be redistributed based on the prefix of some hash values (as done in [101]). Another strategy is performed by [19]. The triples are sorted lexicographically and one half of them is sent to another compute node. In both cases the systems keep track to which compute node they have moved the triples.

5 Indices

RDF stores in the cloud distributed the triples of an RDF graph over several computers. When a query is sent to these RDF stores, they require an index which can tell on which compute nodes the data contributing to the query processing is located. These indices are either stored on a single compute node – i.e., the master or the query federator – (see section 5.1) or they are distributed over several compute nodes (see section 5.2). A centralized index has the advantage that it has knowledge about all graph chunks. To reduce the size of the index, some type of aggregation needs to be applied. In contrast to this distributed indices need fewer aggregation, since they are stored across all compute nodes. But a single index lookup might require the routing of the lookup via several compute nodes until the required information is found.

5.1 Centralized Indices

Hash-Based Index. In distributed RDF stores that apply some variant of a hash cover strategy (for instance, Virtuoso Clustered Edition [40], 4store [58] or Trinity.RDF [145]), no explicit index is required. Based on the knowledge of all compute nodes, the hash function and the triple elements that were hashed, the compute node to which triples following a specific pattern were assigned can be identified.

Statistics-Based Index. Another type of centralized indices base on statistical information about the resources occurring in the individual graph chunks. In RDF stores like DARQ [115], FedX, [133] and Sedge [142] the frequency of every subject, property and object in each chunk is counted and stored. Since these information do not tell anything about the RDFS types contained in the graph chunks, systems like SPLENDID [115], WoDQA [115], LHD [115], SemaGrow [115], Avalanche [115] bases on VoID descriptions [11] of each graph chunk. These descriptions contain the occurences of URIs in the dataset, the used RDFs types and the properties that occur in triples whose objects occur as subject in triples assigned to a different compute node. Since these information might be complicated to collect in a federated setting, if the remote RDF stores do not provide VoID descriptions, ANAPSID [115] restricts itself to only count the occurrences of properties and RDFS types.

If a triple pattern with two constants is requested, the indices described so far could only restrict the number of queried compute nodes by either of the two constants. To restrict the number of queried compute nodes even further, LILAC [92], SemStore [138] additionally count how frequently all subject-property, property-object and subject-object combinations occur.

Since not all subjects and objects occurring in a dataset have an RDFS type, the RDF store Odyssey [90] defines the type of an subject v by the set of properties occurring in its molecule M_v. Since the number of types might me very large, types with similar property sets are merged. Additionally, it counts how frequently instances of molecule types are connected by properties.

Summary-Graph-Based Index. In distributed RDF stores summary graphs are created. This summary graph can be queried to identify compute nodes that store triples required for the processing of a query. The definition of a summary graph differ between the RDF stores.

In TriAD [52] each graph chunk becomes a vertex in the summary graph. Since TriAD uses minimal edge-cut cover, the underlying algorithm assigns each vertex to a compute node. For each triple (s, p, o), an edge with label p is crated in the summary graph that connects the vertices representing the chunks stored on the compute nodes to which s and o were assigned to. To reduce the size of the summary graph, all edges connecting the same vertices and having the same label are represented by only a single edge.

Example 13. Figure 12 shows the summary graph created from the minimal edge cut cover in Figure 10. The vertices represent the compute nodes. For instance, c_1 represents the graph chunk of compute node c_1. The self-loop at the vertices represent the

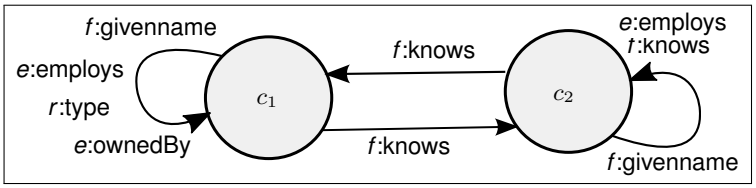


Fig. 12: The summary graph of the graph cover shown in Figure 10 used by TriAD.

properties occurring within these compute nodes. To simplify the graphic, all labels were attached to a single edge instead of creating an own edge for every label. The two edges connecting both vertices represent both triples whose subject and object were assigned to different compute nodes.

Another type of summary graph is used by EAGRE [146]. Vertices in the summery graph represent molecule types. A molecule type is a set of all properties that occur in at least one molecule. To reduce the number of molecule types, molecule types with similar properties are merged. Each triple t contained in a molecule of type T_1 whose object is the anchor vertex of another molecule of type T_2 will result in an edge with the label of the property that connects the vertices T_1 and T_2 of the summary graph.

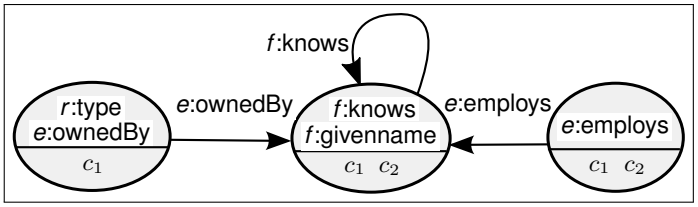


Fig. 13: The summary graph of the graph cover shown in Figure 10 used by EAGRE.

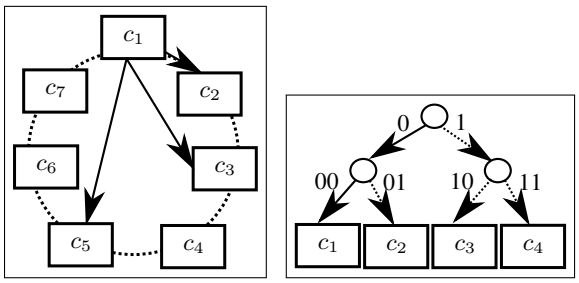
Example 14. Figure 13 shows the summary graph created by EAGRE for the graph cover shown in Figure 10. The three vertices represent the three different molecule types. The properties occurring in molecules of that type are written in the upper part of the vertex. The compute nodes on which molecules of that type can be found are listed in the bottom part of each vertex. The leftmost vertex represents the dog molecule, the middle vertex represents the employee molecules and the right vertex represents the institutes molecules. The e:employs edge connecting the institutes molecule type with the employees molecule type represents the edges that connect the two institutes with their employees.

5.2 Distributed Indices

In distributed indices every compute node knows only a part of the complete index. In order to find every entry of the complete index, the compute nodes have to forward

the index lookup request to other compute nodes, until the compute node knowing the requested information is found. In order to route these index lookups, overlay networks are created that define to which compute node an index lookup should be forwarded.

Hash-Based Index. If a distributed RDF store uses a hash partitioning, each compute node can determine on which compute node a triple can be found, by knowing the set of all available compute nodes. This approach is done, for instance, by HDRS [22] and Virtuoso Clustered Edition [41].



(a) Ring overlay. (b) Tree overlay.

Fig. 14: The different types of overlay networks used in distributed hash indices.

In peer-to-peer distributed RDF stores, the set of all compute nodes might be large and change over time. To prevent the replication of this set and keeping it up-to-date on all compute nodes, each compute node only stores a set of neighboured compute nodes, to which index lookups might be forwarded. The definition of the neighbourhood creates an overlay structure. In peer-to-peer distributed RDF stores the following overlay structures are used frequently:

Ring structure. In RDFPeers [25], PAGE [35], Atlas [67] and [82] the compute nodes are order, e.g., by their IP address. This order defines the direct neighbours of each compute node. To ensure that every compute node has exactly two neighbours, the first and last compute node are defined as neighbours. An example of the resulting ring is shown in Figure 14a. If only the ring structure would be given, finding a compute node that stores the requested data would take linear time. To reduce the lookup time, each compute nodes stores shortcuts to compute nodes at later positions in the ring. For instance, compute node c_1 knows compute nodes c_2, c_3 and c_5. If c_1 is asked whether it knows some information about resource r, it can compute with the hash that triples with this resource would have been assigned to, e.g., c_6. Since he does not know c_6 he sends the request to c_5 which is closest to c_6.

Tree structure. In GridVine [6, 31], UniStore [68], 3RDF [12], [14], [13] the overlay network is based on a prefix tree as shown in Figure 14b. Each vertex in this tree represents a prefix. The root has an empty prefix, the left child of the root node has the prefix 0 and the leaf c_1 stores all triples with resources whose hash value start

with 00. Each compute node knows the path from the root to itself. For each node n in the path, the compute node knows one compute node contained in the subtree of the siblings of n. For instance c_1, would forward every hash with prefix 01 to compute node c_2 and every hash with prefix 1 to compute node c_3.

Combinations of both overlay structures can be found in [19] and [101]. The basic idea is initially they use the ring overlay structure. If one compute nodes has to store too many entries, it redistributes it triples based on a tree structure.

Schema-Based Index. Instead of using hash-based indices, the RDFS types can be used to distribute data. The RDFS types contained in a dataset usually build a type hierarchy. Similar to the tree overlay structure presented above, each compute node is responsible for the instances of the RDFS types assigned to him. If a compute node should retrieve instances of a given type T that is not assigned to him, it searches for a superclass of T for which he knows a responsible compute node and forwards the request to it. This so called semantic overlay network [30] is used, for instance, by SQPeer [73].

Chunk-Integrated Summary Graph Index. A completely different type of distributed index is presented in [108, 109]. It adapts the idea of the summary graph from TriAD as described in section 5.1. Instead of realizing an separate index structure, it integrates the summary graph into its local RDF storage. With the help of these additional information a compute node can decide, whether the triples of another compute node has data that might lead to further query results.

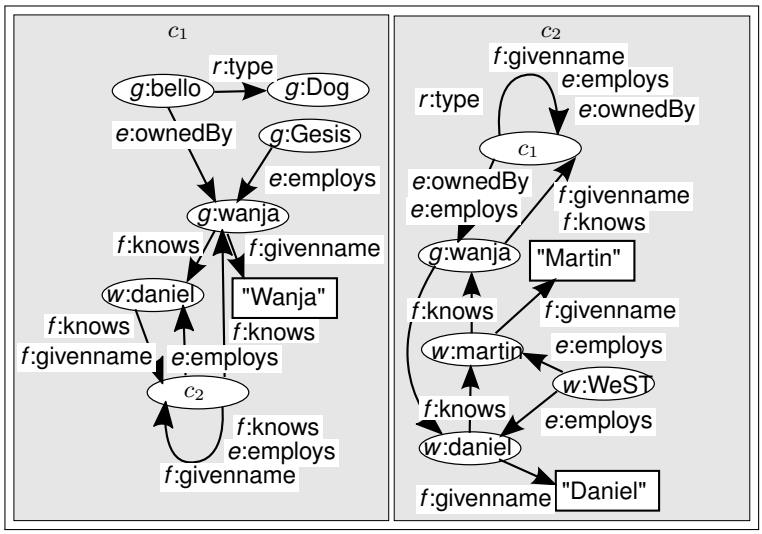


Fig. 15: The summary graph integrated into the graph cover shown in Figure 10.

Example 15. Figure 15 shows the minimal edge-cut graph cover from Figure 10 extended by the triples from the summary graph. On c_1 the graph chunk from c_2 is represented by a super vertex named c_2. All triples whose subject or object is contained in the graph chunk of c_1 but the counterpart not, is represented by an edge connecting the vertex within the chunk of c_1 with the super vertex representing the graph chunk in which the other vertex is contained. For instance, the subject of the triple `w:daniel f:knows w:martin` is contained in the chunk of c_1 whereas its object is only contained in the chunk of c_2. Therefore, the chunk of c_1 is extended by the triple `w:daniel f:knows` c_2. Furthermore, for each property p label occurring in the graph chunk of c_2, a triple c_2 p c_2 is added to the chunk of compute node c_1.

6 Distributed Query Processing Strategies

RDF stores in the cloud distribute RDF graphs over several compute nodes. One challenge which arises from this distribution is how to query the distributed graph. In general RDF stores in the cloud try to compute as much on the graph chunks as possible without the need to exchange data. Therefore, the received query is decomposed into subqueries that can be executed only on the individual graph chunks stored on a single compute node. In the simplest case these subqueries consist of a single triple pattern. Other systems can make use of some properties of the underlying graph cover. For instance, SHARD [103] uses a graph cover that assigns all triples with the same subject to one compute node. As a result, all star-shaped subqueries can be executed locally. Another example are RDF stores that make use of the n-hop replication. This replication ensures that all queries that match with a subgraph with diameter n can be executed locally. If the local RDF storage is able to return all partial matches of the query, the complete query can be executed by the local RDF storage. With the help of the indices, the number of compute nodes that can contributed to a subquery can be restricted.

The intermediate results of the subqueries needs to be joined in order to produce the overall query result. Several RDF stores transfer all intermediate results to a single compute node that is then responsible for joining it (see section 6.1). If a huge number of intermediate results are produced, the joining compute node might be overloaded. Therefore, several RDF stores distribute the join computation over several compute nodes (see section 6.2). An example how queries are executed in distributed graph processing frameworks is given in section 6.3.

6.1 Centralized Join

Especially in federated RDF stores, the intermediate results of subqueries that were executed on remote RDF stores have to be joined on the query federator. Thereby, the join strategies of relational databases are applied. The following join strategies are used:

Nested loop join [89]: For each element in the intermediate result list of the first subquery, the intermediate result list of the second result list needs to be iterated completely to find all join candidates.

Merge join [89]: For this join the intermediate result lists must be ordered. One list is iterated and for each element the join candidates in the other list can be retrieved. Due to the ordering the other list does not need to be traversed from the beginning. Instead only the elements with the same value of the join variable needs to be reiterated. This join is performed by the distributed RDF store Partout [43].

Hash join [36]: The intermediate results of the subqueries are stored in separate hash tables. Each hash table distributes the results into several buckets. If both hash tables use the same number of buckets, only the intermediate results of two buckets need to be joined at once. When all buckets are joined, the join is finished.

Symmetric join [136]: This type of join is a non-blocking hash join. For every subquery a hash table is created that stores the already received intermediate results. When a new intermediate result is received, it is joined with all join candidates in the other hash tables. The results are emitted and the intermediate result is inserted in the hash table of the subquery that produced it. This join strategy is performed by, e.g., ANAPSID [7] and LHD [134].

Bind join [54]: In order to perform a bind join, the first subquery is executed. For each returned distinct intermediate variable binding μ, the second subquery is executed. Thereby, all variables bound by μ in the second subquery are substituted by the bound values. This type of join is performed by, e.g., FedX [127], Avalanche [18] and SemaGrow [26].

Beside performing a single type of join operation, RDF stores like DARQ [115] and SPLENDID [48] choose between a bind join and a nested loop join or between a bind join and a hash join, respectively. The choice depends on the expected number of returned results.

Another join strategy is performed by [105]. In this distributed RDF store partial evaluation [65] is performed. This means that the complete query is executed on the local RDF stores of each compute node. These local RDF stores return the overall results and all intermediate results. For each intermediate result the subquery that created the result is returned. The intermediate results from all compute nodes are sent to a single compute node who finally joins the intermediate results and returns the overall results.

6.2 Decentralized Join

The subqueries in which a query is decomposed can easily produce a large number of intermediate results. Joining all intermediate results on a single compute node can overload the capacity of this compute node. To overcome this limitation several RDF stores in the cloud apply distributed joins.

Replication-Based Distributed Join. In order to distribute the number of join computations over the individual compute nodes, in SemStore [138] all intermediate results of a subquery are sent to all compute nodes on which the succeeding subquery is executed. This strategy increase the amount of transferred intermediate results but each compute node only joins its local results with the intermediate results produced by the other compute nodes.

Distributed Hash Join. In DiploCoud [139] the intermediate results of the subqueries are joined by a distributed hash join. Basically, the distributed hash join is similar to a centralized hash join. The only difference is that each compute node is a bucket of the used hash table.

The hypercube hash join was initially presented in [20] and was used in the distributed RDF store presented in [28]. The basic idea is that for each join variable one dimension is created. For instance, if a query has three join variable, three dimensions are created. Therefore, we need to arrange the compute nodes as a three-dimensional cube. Each compute node is responsible for one cubic region within in this cube. Thereafter, every triple pattern is executed in parallel producing variable bindings. If the variable binding $\{(\texttt{?v1}, \texttt{w:martin})\}$ is produced, it is forwarded to all compute nodes that are responsible for the value $\mathrm{hash}(\texttt{w:martin})$ in the `?v1`-dimension. Each compute nodes performs a local join of the variable bindings it has received from the different triple patterns. Depending on the selection of the regions each compute node is responsible, the workload can be equally distributed among all compute nodes.

Distributed Merge Join. In a distributed merge join the intermediate result lists of the subqueries are sorted by the values of the join variables. Thereafter, each compute node receives all elements within a specific value range and joins them. This type of join is primarily performed in Hadoop-based RDF stores like H2RDF+ [103], SHARD [103], [63] and Spark-based RDF stores like SparkRDF [141] and SPARQLGX [50].

Distributed Bind Join. On way to realize a distributed bind join is implemented in AdHash [9]. The first triple pattern is executed on each compute node. Thereafter, each compute node performs a centralized bind join based on the variable binding the first triple pattern has produced on this compute node.

In RDFPeers [25], GridVine [6, 31], Atlas [67] and 3RDF [12] a hash cover is applied which each triple is stored on at most three compute nodes based on the hash of its subject, property and object. As a consequence all triples in which one resource occurs are located on the same compute node. Since usually each triple pattern of a query contains at least one constant, all matches for this triple pattern can be found on a single compute node. In order to process a query, the query coordinator determines a sequence in which the compute nodes should be traversed to process all triple patterns. When moving from one compute node to the other, all intermediate results are transferred. In order to join the intermediate results, bind joins are performed.

In order to generalize and parallelize the previously described strategy, [82] introduced the so called spread by value querying strategy[26]. The basic idea is that the first triple pattern is processed on the computed nodes on which matches occur. These compute nodes start a bind join processing with the second triple pattern. When a compute node identifies with the help of the global index that for one triple pattern there exist matches on a different compute node, it will fork the query processing on the other compute node that will continue with this branch of the query execution. The final

[26] This idea is named differently in the literature. For instance, in Trinity.RDF [145] it is called graph exploration.

query results are sent back to the query coordinator. This strategy was also used by, for instance, [14], [13], [108, 109] and TripleRush [130]. In order to speed up the query execution, Trinity.RDF [145] performs the spread by value strategy from the first and the last triple pattern in parallel.

Example 16. In this example the basic graph pattern `<w:WeST> <e:employs> ?v1. ?v1 <f:knows> ?v2. ?v2 <f:givenname> ?v3` should be executed on the graph cover with integrated summary graph index in figure 15. The first triple pattern creates the variable binding $\{(\texttt{?v1}, \texttt{w:martin})\}$ on compute node c_2. Based on this variable binding the variable `?v1` of the second triple pattern will be substituted by `w:martin`. When processing the triple pattern `<w:martin> <f:knows> ?v2`, the variable binding $\{(\texttt{?v1}, \texttt{w:martin}), (\texttt{?v2}, \texttt{g:wanja})\}$ is produced. Before processing the third triple pattern, `?v3` will be substituted by `g:wanja`. As a result `<g:wanja> <f:givenname> ?v3` is executed. This time the only possible substitution for `?v3` is the super vertex c_1. This match means that there exists a triple on compute node c_1 that would match with the triple pattern. Therefore, the query, the created variable binding, and the metadata that this variable binding was created by the first two triple pattern is sent to compute node c_1. Now, c_1 executes `<g:wanja> <f:givenname> ?v3` and produces the resulting variable binding $\{(\texttt{?v1}, \texttt{w:martin}), (\texttt{?v2}, \texttt{g:wanja}), (\texttt{?v3}, \texttt{"Wanja"})\}$. For the sake of simplicity, the other intermediate results produced by the triple patterns that were executed in parallel were skipped during the explanation of this example.

6.3 Distributed Query Processing in Graph Processing Frameworks.

In S2X [141] a different type of distributed query processing is presented. First, each vertex checks whether it can be a substitution for some variable in the query by checking its incident edges, their labels and the adjacent vertices. Thereafter, it notifies the neighboured vertices by its variable bindings. If for one variable binding no join compatible variable binding can be found on the neighboured vertices, it is discarded. The notification of the neighboured vertices and the discard of local variable bindings is repeated until the variable binding of each vertex in the graph does not change any more. The remaining variable binding can be retrieved and joined as the final results.

7 Fault Tolerance

One problem of RDF stores in the cloud is that a single compute node might fail or become disconnected from the network. RDF stores that bases on cloud computing frameworks mainly rely on the fault tolerance of the used framework. In federated RDF stores the actual data is stored on remote RDF stores that are not under control of the system administrator. As a result the fault tolerance is not an urgent problem for both types of RDF stores.

In contrast to these RDF stores, the failure of compute nodes is an issue for distributed RDF stores. Since most of these RDF stores that can be found in the literature are proof-of-concept implementations, they do not address the problem of fault tolerance. The few systems that deal with fault tolerance, address this problem by replication. Systems like Virtuoso Clustered Edition [41] suggest to create identical copies of all compute nodes. If one compute node fails, it is replaced by one of its copies.

Another strategy to become fault tolerant is used by, e.g., 4store [58] and RDFPeers [25]. In these distributed RDF stores there exists an partial order of all compute nodes, for instance, created by the comparison of their IP addresses. To ensures that every compute node has a successor and predecessor, the successor of the last compute node is the first compute node. Based on this ordering, the triples assigned to one compute node are also assigned to the neighboured compute nodes. If one compute node fails, the index forwards the queries to one of neighbours that store replicas of the data originally assigned to the failed compute node.

8 Evaluation Methodologies

In order to evaluate the performance of RDF stores several benchmarks were proposed. In general benchmarks consist of a dataset, a set of queries and several performance metrics. In order to test the RDF stores with differently-sized RDF graphs, benchmarks usually use a dataset generator that generated RDF stores based on a schema and/or specific characteristics. Some benchmarks provide fixed queries or query patterns that contain special variables that are substituted by constants after dataset generation (see section 8.1). Instead of providing query patterns, benchmark generators generate queries based on query characteristics (see section 8.2). In section 8.3, we elaborate how benchmark are used to evaluate distributed RDF stores.

8.1 Benchmarks

Lehigh University Benchmark. LUBM [51] was developed to test the query optimizer performance. It generates an RDF graph based on its Univ-Bench ontology. This ontology describes universities, their departments, employees, courses, students and related activities. In order to provide more realistic datasets, several constraints are applied during data generation. For instance, a university can have between 15 and 25 departments

Query	# Triple Patterns	Query Diameter
Q1	2	1
Q2	6	2
Q3	2	1
Q4	5	1
Q5	2	1
Q6	1	1
Q7	4	2
Q8	5	2
Q9	6	2
Q10	2	1
Q11	2	1
Q12	4	2
Q13	2	2
Q14	1	1

Table 1: LUBM query characteristics.

and the ratio between undergraduate students and faculty is between 8 and 14. The 14 provided SPARQL queries are designed to test how well the query optimizer can improve the join ordering. The characteristics of the queries are shown in table 1. LUBM proposes the following performance metrics:

Load time is the time that the RDF store needs to parse and load the RDF graph.

Repository size is the size of all files that are required by the RDF store to store the dataset including dictionary and indices.

Query execution time is the average time to execute a query ten times.

Query completeness and soundness measures with percentage of all results were retrieved and the percentage of correct results.

SP²Bench SP²Bench [126] was designed to test the most common SPARQL constructs and how they are applied in realistic queries. It provides a dataset generator that creates an RDF graph that follows the DBLP schema. This schema describes publications like articles and inproceedings with their bibliographic information. The generated graph mimics the characteristics of the real DBLP graph. SP_2Bench provides 17 queries. They mainly focus on testing the join ordering capabilities of the query optimizer but also complex filters and duplicate elimination. The characteristics of the queries are given in table 2. The proposed performance metrics are:

Load time is the time that the RDF store needs to parse and load the RDF graph.

Query execution time is the average time to execute a query.

Global query execution time is the arithmetic and geometric mean of all 17 queries. It is computed by multiplying the execution times of all 17 queries and then computing the 17th root of the result. If a query could not be processed, it is punished with 3600 seconds.

Memory consumption is measured by (i) the maximum amount of memory that was allocated during the processing of each individual query and (ii) the average memory consumption of all queries.

Berlin SPARQL Benchmark (BSBM). The BSBM[27] [21] focusses on the use case of an e-commerce platform. It aims to simulate the search and navigation patterns of multiple concurrently acting customers. The dataset is generated based on a relational schema. This schema represents products, their offers and the custom reviews of the products. BSBM provides 12 query patterns whose characteristics are given in table 3. These query mainly test the ability to optimize the join ordering and the early application of filters. In BSBM a query pattern refers to a query in which some constants are replaced by a special type of variable. During the runtime of the benchmark these special variable are replaced with varying constants occurring in the dataset. A set of queries in which each query pattern is instantiated is called a query mix. Several of these query mixes are executed in parallel in order to measure the performance of the RDF store. The proposed performance metrics are:

[27] `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/`

Query	# Triple Patterns	Query Diameter
Q1	5	1
Q2	10	1
Q3a	2	1
Q3b	2	1
Q3c	2	1
Q4	8	2
Q5a	6	2
Q5b	6	2
Q6	9	2
Q7	13	5
Q8	8	2
Q9	4	2
Q10	1	1
Q11	1	1
Q12a	6	2
Q12b	8	2
Q12c	1	1

Table 2: SP^2Bench query characteristics.

Load time is the time that the RDF store needs to parse and load the RDF graph.

Query mixes per hour is the number of query mixes that can be completely processed within one hour.

Queries per second is the number of queries, which are instantiated from a single query pattern, that can be answered within one second.

Query	# Triple Patterns	Query Diameter
Q1	5	1
Q2	15	2
Q3	7	1
Q4	10	1
Q5	7	1
Q6	2	1
Q7	14	2
Q8	10	2
Q9	1	1
Q10	7	2
Q11	2	2
Q12	9	2

Table 3: BSBM query characteristics.

Query	# Triple Patterns	Query Diameter
Q1	26	3
Q2	9	2
Q3	8	1
Q4	4	1
Q5	6	2
Q6	5	2
Q7	6	1
Q8	11	2
Q9	9	1
Q10	7	3
Q11	8	2

(a) Basic query set characteristics.

Query	# Triple Patterns	Query Diameter
Q1	26	3
Q2	9	2
Q3	1	1
Q4	3	1
Q5	3	2
Q6	4	2
Q7	2	1
Q8	4	1
Q9	1	1
Q10	5	1
Q11	11	1
Q12	9	1
Q13	5	1
Q14	12	2
Q15	10	2
Q16	9	1
Q17	5	2
Q18	6	1
Q19	4	1
Q20	11	2
Q21	9	1
Q22	9	1
Q23	9	1
Q24	4	1
Q25	4	1

(b) Advanced query set characteristics.

Table 4: SPB query characteristics.

Semantic Publishing Benchmark (SPB). The SPB[28] [74] is a benchmark motivated by the industry. The use case is a publisher organization that provides metadata about its published work. Many journalists search for data and perform insertions and deletions concurrently. SPB provides a dataset generator that is designed to create datasets with several billions of triples that mimic the BBC datasets. Similar to BSBM it provides query templates that contains special variables that will be instantiated before query execution. SPB defines two set of query templates. The basic query set focuses on join ordering, duplicate elimination and filtering whereas the advanced query set additionally contains, e.g., analytical queries. The query characteristics are given in table 4a and in table 4b. The proposed performance metrics are:

Minimum, maximum and average query execution time for each executed query.

Average execution rate per second measures how many queries could be finished per second in average.

[28] `http://ldbcouncil.org/developer/spb`

FedBench. FedBench [125] is designed as a benchmark for federated RDF stores. It provides three different dataset collections: (i) a general linked data collection containing DBPedia, GeoNames, Jamendo, Linked_MDB, New York Times and Semantic Web Dog Food, (ii) a life science data collection containing KEGG, ChEBI and DrugBank as well as (iii) a dataset of 10M triples generated with the dataset generator of SP^2Bench. FedBench provides two self-made query sets as well as the queries from SP^2Bench for the three data collections. The first two query sets focus on the number of data sources involved, the join ordering and query results set sizes. Since the actual benchmark cannot be found online any more, the characteristics of the queries cannot be examined. The proposed performance metrics are:

Query execution time for each executed query.
Number of requests to remote RDF stores during the processing of each query.

8.2 Benchmark Generators

DBPedia SPARQL Benchmark (DBSB). The general idea of DBSB [94] is to scale the DBPedia dataset to the required size and create queries based on a historic query log of DBPedia SPARQL endpoints. In order to generate the dataset a DBPedia dump is taken. To increase the size, triples are replicated and there namespaces are changed. To shrink the dataset size, triples are removed in a way that its characteristics like the indegree and the outdegree of vertices is not changed. In order to generate queries, DBSB clusters all queries of a historic query log. Out of each cluster the most frequent queries were picked as well as queries that cover most SPARQL features. Based on the selected queries, new queries are generated by replacing the constants with resources of the generated dataset during the benchmark generation process.

Waterloo SPARQL Diversity Test Suite (WatDiv). WatDiv [15] was designed to create benchmarks that are able to test the performance change of RDF stores under varying dataset and query characteristics. Therefore, the dataset generator is able to create datasets with variations of (a) the entity types, (b) the graph topology, (c) the well-structuredness of entities (i.e., which portion of the defined edges are usually present at an entity), (d) the probability of edges connecting two entities and (e) the cardinality of properties. In order to generate queries based on a dataset, the following characteristics are defined:

Triple Pattern Count defines the number of triple patterns occurring in the generated query.
Join Vertex Count counts the number of resources or variables that occur in multiple triple patterns.
Join Vertex Degree determined in how many triple patterns each join vertex occurs.
Join Vertex Type defines whether a subject-subject, subject-object or object-object join is performed.
Result Cardinality is the number of results.
Filter Triple Pattern Selectivity defines with witch portion of the graph a triple pattern matches.

BGP-Restricted f-TP Selectivity determines to which extent a triple pattern contributes to the overall selectivity of a query.

Join-Restricted f-TP Selectivity determines to which extent a triple pattern contributes to the overall selectivity of a join.

FEASIBLE. FEASIBLE [119] does not provide a dataset generator. Instead it can use an arbitrary dataset for which a historic query log exists. FEASIBLE aims to generate queries that have similar characteristics to the queries in the query log. Therefore, in a first step all syntactical incorrect queries and queries with no results are removed. Each query is transformed into a vector based on the following query characteristics:

SPARQL features defines which SPARQL features like `SELECT`, `ASK`, `UNION`, etc. occur in the query.

Triple Pattern Count defines the number of triple patterns occurring in the generated query.

Join Vertex Count counts the number of resources or variables that occur in multiple triple patterns.

Join Vertex Degree determined in how many triple patterns each join vertex occurs.

Join Vertex Type defines whether a subject-subject, subject-object or object-object join is performed.

Triple Pattern Selectivity defines with witch portion of the graph a triple pattern matches.

From the resulting set of query vectors, the requested number of queries are selected in a way that their vectors are as far away as possible from each other.

SPLODGE. The idea of SPLODGE [47] is to generated queries with a given set of characteristics from an arbitrary dataset. Thereby, it uses the following query characteristics:

Query Type defines whether a `SELECT`, `CONSTRUCT`, `ASK` or `DESCRIBE` query should be generated.

Join Type defines whether a conjunctive join (`.`), disjunctive join (`UNION`) or left-join (`OPTIONAL`) should be performed.

Result Modifiers defines whether the result set should be altered by `DISTINCT`, `LIMIT`, `OFFSET` or `ORDER_BY` operators.

Variable Patterns defines at which positions of the triple pattern variables should occur.

Join Patterns defines whether a subject-subject, subject-object or object-object join is performed.

Cross Products defines whether conjunctive join without join variable should be performed.

Number of Sources defines from how many different data sources triples should be combined to answer the query.

Number of Joins defines how many joins should occur in the query.

Query Selectivity defines with witch portion of the graph all triple patterns of a query match.

8.3 Performed Evaluations

The before mentioned benchmarks are usually used to evaluate and compare the performance of RDF stores as a whole. Table 5 summarizes the evaluations published from the beginning of 2016. All of them use are least one of the benchmarks described above. Beside the generated datasets they usually also use a few realistic datasets. The maximal dataset size is in most cases approximately 1 billion triples. In two cases a dataset with up to 4.2 billion triples was used. Most RDF stores in the cloud were deployed on 10 compute nodes. In one evaluation 19 compute nodes were used.

Paper	Benchmark	Max. Dataset Size	# compute nodes	compute node size
[29]	WatDiv	~100M triples	10	6 cores, 32 GB RAM, 4 TB disk
[112]	WatDiv	~1,000M triples	10	6 cores, 32 GB RAM, 4 TB disk
[96]	LUBM	~1,330M triples	18	12 cores, 50 GB RAM
	WatDiv			
[92]	WatDiv	~10M triples	11	4 cores, 24 GB RAM
[86]	LUBM	~35M triples	18 slaves	16 cores, 28 GB RAM
			1 federator	16 cores, 56 GB RAM
[5]	LUBM	~4,200M triples	12	24 cores, 148 GB RAM
[140]	LUBM	~220M triples	4-16 slaves	4 cores, 8 GB RAM, 500 GB disk
			1 master	4 cores, 16 GB RAM, 500 GB disk
[124]	WatDiv	~1,000M triples	10	6 cores, 32 GB RAM, 4 TB disk
[121]	WatDiv	~100M triples	10	6 cores, 32 GB RAM, 4 TB disk
[109]	WatDiv	~1,382M triples	10	8 cores, 32 GB RAM
	LUBM			
[106]	BSBM	~5M triples	4-12	2 cores, 8 GB RAM
[105]	WatDiv	~1,099M triples	10	4 cores, 16 GB RAM, 500 GB disk
	LUBM			
	FedBench			
[104]	WatDiv	~250M triples	10	4 cores, 16 GB RAM, 150 GB disk
[57]	WatDiv	~4,288M triples	5-12	24 cores, 148 GB RAM
	LUBM			
[50]	WatDiv	~1,380M triples	10	4 cores, 17 GB RAM
	LUBM			
[4]	LUBM	~3,100M triples	11	8 cores, 16 GB RAM, 3TB disk

Table 5: Evaluations of RDF stores in the cloud published since 2016.

The evaluations in these papers use rather small datasets. To give a better overview of the capabilities of current RDF stores, [1] reports RDF stores running on a single server or in the cloud that could store RDF graphs consisting of several billions or even one trillion triples (see the summary in table 6).

[29] `http://www.oracle.com/us/corporate/features/database-12c/index.html`

[30] `https://franz.com/agraph/allegrograph/`

[31] `https://www.stardog.com/`

RDF Store	Max. Dataset Size	# compute nodes	compute node size
Oracle Database 12c[29]	~1 trillion triples	1	360 cores, 2 TB RAM, 45 TB disk
AllegroGraph[30]	~1 trillion triples	1	?
Stardog[31]	~50,000M triples	1	32 cores, 256 GB RAM
Virtuoso Clustered Edition [41]	~37,000M triples	8	8 cores, 16 GB RAM, 4 TB disk
GraphDB[32]	~17,000M triples	1	16 cores, 512 GB RAM
4store [58]	~15,000M triples	9	?
Blazegraph [2]	~12,700M triples	?	?
YARS2 [2]	~7,000M triples	?	?
Jena TDB[33]	~1,700M triples	1	2 cores, 10 GB RAM
RDFox[34]	~1,700M triples	1	16 cores, 50 GB RAM

Table 6: Evaluations of RDF stores reported by [1].

In order to compare the influence of alternative graph cover strategies or different query execution strategies, all but the examined component of the distributed RDF store need to stay the same. This was done, for instance in [96] to compare different query execution strategies on top of Spark or in [33], [63], [79] and [146] to compare different graph cover strategies. But these evaluation used Hadoop or its distributed file system to exchange data during the query processing. As a result, it remains unclear whether their results are applicable on distributed RDF stores in which the data is exchanged directly between the compute nodes.

9 Conclusion

To cope with the growing size of huge graphs, scalable RDF stores in the cloud are used, where the graph data is distributed among several compute nodes. From this distributed setting several challenges like (i) the data placement strategy, (ii) the distributed query processing, and (iii) the handling of failed compute nodes. In this manuscript we gave an overview of how these challenges are addressed by RDF stores in the cloud.

Due to the high number of RDF stores in the cloud, we have only given an overview of core challenges in distributed RDF stores. Beside these core challenges there exist further features that are required during the practical usage of relational databases in the industry today. Realizing them in RDF stores is a challenging task so that they are only partly realized in RDF stores. In order to achieve a broader usage of RDF stores in industry, further research is required to implement these features in RDF stores in the cloud. In the following we describe some of these features. As an example use case we assume an online retailer.

When two customers try to order a unique product at the same time, only one of the orders must be successful and the other must fail. To prevent the situation that both cus-

[32] `www.ontotext.com/products/ontotext-graphdb/`
[33] `https://jena.apache.org/`
[34] `http://www.cs.ox.ac.uk/isg/tools/RDFox/`

tomers could successfully order the unique product, *transactional security* (i.e., atomicity, consistency, isolation and durability) is required. Realizing transactional security in a distributed setting where the data is separated among several compute nodes might require a lot of coordination between the compute nodes. This additional coordination increase the query query execution time. To avoid this overhead while providing transactional security, most RDF stores in the cloud assume that the RDF graph is immutable after loading it. Only few RDF stores like Virtuoso Clustered Edition [40] allow for inserting or deleting triples after loading.

In order to identify which products were sold the most frequently in the last three month, the database is required to perform *online analytical processing (OLAP) queries*. This type of queries require a huge amount of data to be processed. In context of RDF stores in the cloud, processing OLAP queries cannot be done by sending all required data to a single compute node since a single compute node might be overloaded by the huge amount of data. Therefore, graph cover strategies and distributed query execution strategies need to be developed that support the parallel processing of OLAP queries with a low number of exchanged network packets.

To simplify the search for a product, the online retailer has categorized its products in a category hierarchy. For instance, an orange lemonade can be categorized as lemonade, soft drink and drink. With the introduction of SPARQL 1.1 [110] *property paths* were introduced that allow for requesting all offered drinks independently of the subcategory they belong to. This type of query differs from the pure graph pattern matching done in SPARQL 1.0 since it can easily require the traversal of long paths. In a distributed setting the traversal of long paths may lead to a high network traffic reducing the query execution time. One challenge arising from these queries is, how to optimize the data placement for these queries. Alternatively to SPARQL, the retailer might want to use other query languages like GraphQL[35].

The retailer wants to prevent teenagers from buying alcohol. Therefore, he stores in database the rules that every customer younger than 20 is a teenager and teenagers should not be allowed to buy alcohol. These rules should be automatically applied to all customers. In order to realize this, RDF stores need the ability to reason about RDF graphs. In this context *reasoning* means inferring logical consequences and checking the consistency of the RDF graph. Usually, reasoning is done during the loading time of the graph and all logical consequences are stored as explicit triples in the graph. The challenge of distributed reasoning is that the reasoning of the complete graph might overload a single compute node. Therefore, distributed reasoning algorithms are required. A few RDF stores in the cloud like MaRVIN [100] and Rya [114] have addressed this challenge. Another problem is, if the RDF graph is mutable after loading. In this case the deletion or insertion of triples might produce inconsistencies, a lot of newly inferred triples need to be inserted or formerly inferred triples need to be removed.

Finally, the retailer wants to advertise summer products like swimwear, portable fans, etc. more prominently if the temperature in the town where the customer lives is high. Therefore, the constantly streamed data from temperature sensors needs to be processed. This quickly arriving stream data cause further challenges for RDF stores,

[35] `https://graphql.org/`

like quickly combining the received data with static data, updating the database frequently or balancing the workload among all compute nodes. CQELS Cloud [78] is one example of a system that processes RDF streams in a distributed fashion.

References

1. Largetriplestores. https://www.w3.org/wiki/LargeTripleStores, accessed: 2018-07-10
2. The bigdata® RDF Database. Retrieved: 29.10.2014, http://www.bigdata.com/whitepapers/bigdata_architecture_whitepaper.pdf
3. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases. pp. 411–422. VLDB '07, VLDB Endowment (2007), http://dl.acm.org/citation.cfm?id=1325851.1325900
4. Abbassi, S., Faiz, R.: RDF-4X: A Scalable Solution for RDF Quads Store in the Cloud. In: Proceedings of the 8th International Conference on Management of Digital EcoSystems. pp. 231–236. MEDES, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/3012071.3012104
5. Abdelaziz, I., Harbi, R., Salihoglu, S., Kalnis, P.: Combining Vertex-Centric Graph Processing with SPARQL for Large-Scale RDF Data Analytics. IEEE Transactions on Parallel and Distributed Systems 28(12), 3374–3388 (2017)
6. Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Van Pelt, T.: GridVine: Building Internet-Scale Semantic Overlay Networks. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) The Semantic Web – ISWC 2004. pp. 107–121. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
7. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) The Semantic Web – ISWC 2011. pp. 18–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
8. Akar, Z., Halaç, T.G., Ekinci, E.E., Dikenelli, O.: Querying the Web of Interlinked Datasets using VOID Descriptions. In: WWW2012 Workshop on Linked Data on the Web, Lyon, France, 16 April, 2012 (2012), http://ceur-ws.org/Vol-937/ldow2012-paper-06.pdf
9. Al-Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., Sahli, M.: Adaptive Partitioning for Very Large RDF Data. CoRR abs/1505.0 (2015), http://arxiv.org/abs/1505.02728
10. Al-Harbi, R., Ebrahim, Y., Kalnis, P.: PHD-Store: An Adaptive SPARQL Engine with Dynamic Partitioning for Distributed RDF Repositories. CoRR abs/1405.4 (2014), http://arxiv.org/abs/1405.4979
11. Alexander, K., Cyganiak, R., Hausenblas, M., Zhao, J.: Describing Linked Datasets with the VoID Vocabulary. W3c interest group note, W3C (2011), http://www.w3.org/TR/2011/NOTE-void-20110303/
12. Ali, L., Janson, T., Lausen, G.: 3rdf: Storing and Querying RDF Data on Top of the 3nuts Overlay Network. In: 2011 22nd International Workshop on Database and Expert Systems Applications. pp. 257–261 (aug 2011)
13. Ali, L., Janson, T., Schindelhauer, C.: Towards Load Balancing and Parallelizing of RDF Query Processing in P2P Based Distributed RDF Data Stores. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 307–311 (feb 2014)

14. Ali, L., Janson, T., Lausen, G., Schindelhauer, C.: Effects of Network Structure Improvement on Distributed RDF Querying. In: Hameurlain, A., Rahayu, W., Taniar, D. (eds.) Data Management in Cloud, Grid and P2P Systems. pp. 63–74. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
15. Aluç, G., Hartig, O., Özsu, M., Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (eds.) The Semantic Web – ISWC 2014, Lecture Notes in Computer Science, vol. 8796, pp. 197–212. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-11964-9_13
16. Arenas, M., Pérez, J.: Federation and Navigation in SPARQL 1.1. In: Eiter, T., Krennwallner, T. (eds.) Reasoning Web. Semantic Technologies for Advanced Query Answering, Lecture Notes in Computer Science, vol. 7487, pp. 78–111. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-33158-9_3
17. Basca, C., Bernstein, A.: Distributed SPARQL Throughput Increase: On the effectiveness of Workload-driven RDF partitioning. In: ISWC2013 (2013)
18. Basca, C., Bernstein, A.: Querying a Messy Web of data with AVALANCHE. Web Semantics: Science, Services and Agents on the World Wide Web 26(0) (2014), http://www.websemanticsjournal.org/index.php/ps/article/view/361
19. Battré, D., Heine, F., Höing, A., Kao, O.: On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT Based RDF Stores. In: Moro, G., Bergamaschi, S., Joseph, S., Morin, J.H., Ouksel, A.M. (eds.) Databases, Information Systems, and Peer-to-Peer Computing. pp. 343–354. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
20. Beame, P., Koutris, P., Suciu, D.: Skew in Parallel Query Processing. In: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. pp. 212–223. PODS '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2594538.2594558
21. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. Int. J. Semantic Web Inf. Syst. 5(2), 1–24 (2009), https://doi.org/10.4018/jswis.2009040101
22. Böhm, C., Hefenbrock, D., Naumann, F.: Scalable Peer-to-peer-based RDF Management. In: Proceedings of the 8th International Conference on Semantic Systems. pp. 165–168. I-SEMANTICS '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2362499.2362523
23. Bröcheler, M., Pugliese, A., Subrahmanian, V.S.: COSI: Cloud Oriented Subgraph Identification in Massive Social Networks. In: Advances in Social Networks Analysis and Mining (ASONAM). pp. 248–255 (Aug 2010)
24. Bugiotti, F., Camacho-Rodríguez, J., Goasdoué, F., Kaoudi, Z., Manolescu, I., Zampetakis, S.: SPARQL Query Processing in the Cloud. In: Harth, A., Hose, K., Schenkel, R. (eds.) Linked Data Management. Emerging Directions in Database Systems and Applications, Chapman and Hall/CRC (Apr 2014)
25. Cai, M., Frank, M.: RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. Proceedings of the 13th International Conference on World Wide Web pp. 650–657 (2004), http://dl.acm.org/citation.cfm?id=988760
26. Charalambidis, A., Troumpoukis, A., Konstantopoulos, S.: SemaGrow: Optimizing Federated SPARQL Queries. In: Proceedings of the 11th International Conference on Semantic Systems. pp. 121–128. SEMANTICS '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2814864.2814886
27. Cheng, L., Kotoulas, S.: Scale-Out Processing of Large RDF Datasets. IEEE Transactions on Big Data 1(4), 138–150 (2015)
28. Chu, S., Balazinska, M., Suciu, D.: From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In: Proceedings of the 2015 ACM SIGMOD Interna-

tional Conference on Management of Data. pp. 63–78. SIGMOD '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2723372.2750545
29. Cossu, M., Färber, M., Lausen, G.: PRoST: Distributed Execution of {SPARQL} Queries Using Mixed Partitioning Strategies. In: Proceedings of the 21th International Conference on Extending Database Technology, {EDBT} 2018, Vienna, Austria, March 26-29, 2018. pp. 469–472 (2018), https://doi.org/10.5441/002/edbt.2018.49
30. Crespo, A., Garcia-Molina, H.: Semantic Overlay Networks for P2P Systems. In: Moro, G., Bergamaschi, S., Aberer, K. (eds.) Agents and Peer-to-Peer Computing. pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
31. Cudre-Mauroux, P., Agarwal, S., Aberer, K.: GridVine: An Infrastructure for Peer Information Management. IEEE Internet Computing 11(5), 36–44 (sep 2007)
32. Cudré-Mauroux, P., Enchev, I., Fundatureanu, S., Groth, P., Haque, A., Harth, A., Keppmann, F., Miranker, D., Sequeda, J., Wylot, M.: Nosql databases for rdf: An empirical evaluation. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) The Semantic Web – ISWC 2013, Lecture Notes in Computer Science, vol. 8219, pp. 310–325. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-41338-4_20
33. Curé, O., Naacke, H., Baazizi, M.A., Amann, B.: On the evaluation of RDF distribution algorithms implemented over apache spark. In: Proc. of the 11th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (at ISWC-2015). pp. 16–31 (2015)
34. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles. pp. 205–220. SOSP '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1294261.1294281
35. Della Valle, E., Turati, A., Ghioni, A.: PAGE: A Distributed Infrastructure for Fostering RDF-Based Interoperability. In: Eliassen, F., Montresor, A. (eds.) Distributed Applications and Interoperable Systems. pp. 347–353. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
36. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D.A.: Implementation Techniques for Main Memory Database Systems. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. pp. 1–8. SIGMOD '84, ACM, New York, NY, USA (1984), http://doi.acm.org/10.1145/602259.602261
37. Dhraief, H., Kemper, A., Nejdl, W., Wiesner, C.: Processing and Optimization of Complex Queries in Schema-Based P2P-Networks. In: Ng, W.S., Ooi, B.C., Ouksel, A.M., Sartori, C. (eds.) Databases, Information Systems, and Peer-to-Peer Computing. pp. 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
38. Ding, L., Peng, Y., da Silva, P.P., McGuinness, D.L.: Tracking RDF Graph Provenance using RDF Molecules. Tech. rep., UMBC (2005), https://ebiquity.umbc.edu/paper/html/id/240/Tracking-RDF-Graph-Provenance-using-RDF-Molecules
39. Du, F., Bian, H., Chen, Y., Du, X.: Efficient SPARQL Query Evaluation in a Database Cluster. IEEE Int. Congress on Big Data pp. 165–172 (2013)
40. Erling, O., Mikhailov, I.: Towards Web Scale RDF. In: 4th Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008) (2008)
41. Erling, O., Mikhailov, I.: Virtuoso: RDF Support in a Native RDBMS. In: de Virgilio, R., Giunchiglia, F., Tanca, L. (eds.) Semantic Web Information Management, pp. 501–519. Springer Berlin Heidelberg (2010)

42. Farhan Husain, M., McGlothlin, J., Masud, M.M., Khan, L., Thuraisingham, B.: Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. Knowledge and Data Engineering, IEEE Transactions on 23(9), 1312–1327 (Sep 2011)
43. Galarraga, L., Hose, K., Schenkel, R.: Partout: A Distributed Engine for Efficient RDF Processing. CoRR abs/1212.5 (2012), http://arxiv.org/abs/1212.5636
44. Goasdoué, F., Kaoudi, Z., Manolescu, I., Quiané-Ruiz, J.A., Zampetakis, S.: CliqueSquare: Flat plans for massively parallel RDF queries. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 771–782 (apr 2015)
45. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: Graph Processing in a Distributed Dataflow Framework. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. pp. 599–613. OSDI'14, USENIX Association, Berkeley, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2685048.2685096
46. Goodman, E.L., Grunwald, D.: Using Vertex-centric Programming Platforms to Implement SPARQL Queries on Large Graphs. In: Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms. pp. 25–32. IA³ '14, IEEE Press, Piscataway, NJ, USA (2014), http://dx.doi.org/10.1109/IA3.2014.10
47. Görlitz, O., Thimm, M., Staab, S.: Splodge: Systematic generation of sparql benchmark queries for linked open data. The Semantic Web–ISWC 2012 pp. 116–132 (2012)
48. Görlitz, O., Staab, S.: SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In: Proceedings of the Second International Conference on Consuming Linked Data - Volume 782. pp. 13–24. COLD'11, CEUR-WS.org, Aachen, Germany, Germany (2010), http://dl.acm.org/citation.cfm?id=2887352.2887354
49. Graux, D., Jachiet, L., Genevès, P., Layaïda, N.: A Multi-Criteria Experimental Ranking of Distributed SPARQL Evaluators (2016), https://hal.inria.fr/hal-01381781
50. Graux, D., Jachiet, L., Genevès, P., Layaïda, N.: SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark. In: Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., Gil, Y. (eds.) The Semantic Web – ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part II, pp. 80–87. Springer International Publishing, Cham (2016), https://doi.org/10.1007/978-3-319-46547-0{_}9
51. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics: Science, Services and Agents on the World Wide Web 3(2-3) (2005), http://www.websemanticsjournal.org/index.php/ps/article/view/70
52. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In: SIGMOD. pp. 289–300 (2014)
53. Gutierrez, C., Hurtado, C., Mendelzon, A.O.: Foundations of Semantic Web Databases. In: PODS. pp. 95–106. ACM (2004)
54. Haas, L.M., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing Queries Across Diverse Data Sources. In: Vldb. VLDB '97, vol. Athens, Gr, pp. 276–285. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997), http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.7366{\&}rep=rep1{\&}type=pdf
55. Hammoud, M., Rabbou, D.A., Nouri, R., Beheshti, S.M.R., Sakr, S.: DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. Proc. VLDB Endow. 8(6), 654–665 (2015), http://dx.doi.org/10.14778/2735703.2735705
56. Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N.: Evaluating SPARQL Queries on Massive RDF Datasets. PVLDB 8(12), 1848–1851 (2015), http://www.vldb.org/pvldb/vol8/p1848-harbi.pdf
57. Harbi, R., Abdelaziz, I., Kalnis, P., Mamoulis, N., Ebrahim, Y., Sahli, M.: Accelerating SPARQL queries by exploiting hash-based locality and adaptive partition-

ing. The VLDB Journal 25(3), 355–380 (jun 2016), https://doi.org/10.1007/s00778-016-0420-y
58. Harris, S., Lamb, N., Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: Scalable Semantic Web Knowledge Base Systems - SSWS2009. pp. 94–109 (2009)
59. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: Proc. of LA-WEB '05. pp. 71—-. IEEE (2005)
60. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: ISWC-2007, vol. 4825, pp. 211–224. Springer Berlin Heidelberg (2007)
61. Hong, S., Depner, S., Manhardt, T., Van Der Lugt, J., Verstraaten, M., Chafi, H.: PGX.D: A Fast Distributed Graph Processing Engine. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 58:1—-58:12. SC '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2807591.2807620
62. Hose, K., Schenkel, R.: WARP: Workload-aware replication and partitioning for RDF. In: Data Engineering Workshops (ICDEW). pp. 1–6 (Apr 2013)
63. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4(11), 1123–1134 (2011)
64. Janke, D., Staab, S., Thimm, M.: Impact analysis of data placement strategies on query efforts in distributed rdf stores. Journal of Web Semantics (2018), http://www.websemanticsjournal.org/index.php/ps/article/view/516
65. Jones, N.D.: An Introduction to Partial Evaluation. ACM Comput. Surv. 28(3), 480–503 (sep 1996), http://doi.acm.org/10.1145/243439.243447
66. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In: 2009 Ninth IEEE International Conference on Data Mining. pp. 229–238 (2009)
67. Kaoudi, Z., Koubarakis, M., Kyzirakos, K., Miliaraki, I., Magiridou, M., Papadakis-Pesaresi, A.: Atlas: Storing, updating and querying RDF(S) data on top of DHTs. Web Semantics: Science, Services and Agents on the World Wide Web 8(4) (2010), http://www.websemanticsjournal.org/index.php/ps/article/view/250
68. Karnstedt, M., Sattler, K.U., Richtarsky, M., Muller, J., Hauswirth, M., Schmidt, R., John, R.: UniStore: Querying a DHT-based Universal Storage. In: 2007 IEEE 23rd International Conference on Data Engineering. pp. 1503–1504 (apr 2007)
69. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. Sci. Comput. 20(1), 359–392 (1998)
70. Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.M., Castagna, P.: Jena-HBase: {A} Distributed, Scalable and Effcient {RDF} Triple Store. Tech. rep., Department of Computer Science at The University of Texas at Dallas (2012)
71. Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.M., Castagna, P.: Jena-HBase: A Distributed, Scalable and Effcient RDF Triple Store. In: Proceedings of the ISWC 2012 Posters & Demonstrations Track, Boston, USA, November 11-15, 2012 (2012), http://ceur-ws.org/Vol-914/paper_14.pdf
72. Kim, H., Ravindra, P., Anyanwu, K.: From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra. PVLDB 4(12), 1426–1429 (2011), http://www.vldb.org/pvldb/vol4/p1426-kim.pdf
73. Kokkinidis, G., Christophides, V.: Semantic Query Routing and Processing in P2P Database Systems: The ICS-FORTH SQPeer Middleware. In: Lindner, W., Mesiti, M., Türker, C., Tzitzikas, Y., Vakali, A.I. (eds.) Current Trends in Database Technology - EDBT 2004 Workshops. pp. 486–495. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

74. Kotsev, V., Kiryakov, A., Fundulaki, I., Alexiev, V.: Ldbc semantic publishing benchmark (spb) - v2.0 first public draft release. Tech. rep., The Linked Data Benchmark Council (June 2014), https://github.com/ldbc/ldbc_spb_bm_2.0/blob/master/doc/LDBC_SPB_v2.0.docx?raw=true
75. Ladwig, G., Harth, A.: CumulusRDF: Linked Data Management on Nested Key-Value Stores. In: Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011) at the 10th International Semantic Web Conference (ISWC2011) (Oct 2011)
76. Ladwig, G., Tran, T.: SIHJoin: Querying Remote and Local Linked Data. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) The Semantic Web: Research and Applications. pp. 139–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
77. Lakshman, A., Malik, P.: Cassandra: A Decentralized Structured Storage System. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (apr 2010), http://doi.acm.org/10.1145/1773912.1773922
78. Le-Phuoc, D., Nguyen Mau Quoc, H., Le Van, C., Hauswirth, M.: Elastic and Scalable Processing of Linked Stream Data in the Cloud. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) The Semantic Web – ISWC 2013. pp. 280–297. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
79. Lee, K., Liu, L.: Efficient Data Partitioning Model for Heterogeneous Graphs in the Cloud. In: Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis. pp. 46:1—-46:12. ACM (2013)
80. Lee, K., Liu, L.: Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. PVLDB 6(14), 1894–1905 (Sep 2013)
81. Lee, K., Liu, L., Tang, Y., Zhang, Q., Zhou, Y.: Efficient and Customizable Data Partitioning Framework for Distributed Big RDF Data Processing in the Cloud. In: IEEE CLOUD '13. pp. 327–334 (2013)
82. Liarou, E., Idreos, S., Koubarakis, M.: Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) The Semantic Web - ISWC 2006. pp. 399–413. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
83. Lynden, S., Kojima, I., Matono, A., Tanimura, Y.: ADERIS: An Adaptive Query Processor for Joining Federated SPARQL Endpoints. In: Meersman, R., Dillon, T., Herrero, P., Kumar, A., Reichert, M., Qing, L., Ooi, B.C., Damiani, E., Schmidt, D.C., White, J., Hauswirth, M., Hitzler, P., Mohania, M. (eds.) On the Move to Meaningful Internet Systems: OTM 2011. pp. 808–817. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
84. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A System for Large-scale Graph Processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. pp. 135–146. SIGMOD '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1807167.1807184
85. Malliaros, F.D., Vazirgiannis, M.: Clustering and community detection in directed networks: A survey. Physics Reports 533(4), 95–142 (2013), http://www.sciencedirect.com/science/article/pii/S0370157313002822
86. Mansour, E., Abdelaziz, I., Ouzzani, M., Aboulnaga, A., Kalnis, P.: A Demonstration of Lusail: Querying Linked Data at Scale. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 1603–1606. SIGMOD '17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3035918.3058731

87. Matono, A., Pahlevi, S.M., Kojima, I.: RDFCube: A P2P-Based Three-Dimensional Index for Structural Joins on Distributed Triple Stores. In: Moro, G., Bergamaschi, S., Joseph, S., Morin, J.H., Ouksel, A.M. (eds.) Databases, Information Systems, and Peer-to-Peer Computing. pp. 323–330. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
88. McMurry, J., Jupp, S., Malone, J., Burdett, T., Jenkinson, A., Parkinson, H., Davies, M., Brandizi, M., et al.: Report on the scalability of semantic web integration in biomedbridges (2015), http://dx.doi.org/10.5281/zenodo.14071
89. Mishra, P., Eich, M.H.: Join Processing in Relational Databases. ACM Comput. Surv. 24(1), 63–113 (1992), http://doi.acm.org/10.1145/128762.128764
90. Montoya, G., Skaf-Molli, H., Hose, K.: The Odyssey Approach for Optimizing Federated {SPARQL} Queries. In: The Semantic Web - {ISWC} 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part {I}. pp. 471–489 (2017), https://doi.org/10.1007/978-3-319-68288-4{_}28
91. Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.E.: Federated SPARQL Queries Processing with Replicated Fragments. In: Arenas, M., Corcho, O., Simperl, E., Strohmaier, M., D'Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thirunarayan, K., Thirunarayan, K., Staab, S. (eds.) The Semantic Web - ISWC 2015, pp. 36–51. Springer International Publishing, Cham (2015)
92. Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.E.: Decomposing Federated Queries in presence of Replicated Fragments. Web Semantics: Science, Services and Agents on the World Wide Web 42(1) (2017), http://www.websemanticsjournal.org/index.php/ps/article/view/486
93. Montoya, G., Vidal, M.E., Acosta, M.: A Heuristic-based Approach for Planning Federated SPARQL Queries. In: Proceedings of the Third International Conference on Consuming Linked Data - Volume 905. pp. 63–74. COLD'12, CEUR-WS.org, Aachen, Germany, Germany (2012), http://dl.acm.org/citation.cfm?id=2887367.2887373
94. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.C.: DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) The Semantic Web – ISWC 2011. pp. 454–469. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
95. Mutharaju, R., Sakr, S., Sala, A., Hitzler, P.: D-SPARQ: Distributed, Scalable and Efficient RDF Query Engine. In: ISWC (Posters & Demos)'13. pp. 261–264 (2013)
96. Naacke, H., Amann, B., Curé, O.: SPARQL Graph Pattern Processing with Apache Spark. In: Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems. pp. 1:1—-1:7. GRADES'17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3078447.3078448
97. Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M., Brunkhorst, I., Löser, A.: Super-peer-based Routing and Clustering Strategies for RDF-based Peer-to-peer Networks. In: Proceedings of the 12th International Conference on World Wide Web. pp. 536–543. WWW '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/775152.775229
98. Norvig, P.: The semantic web and the semantics of the web: Where does meaning come from? In: Proceedings of the 25th International Conference on World Wide Web. pp. 1–1. WWW '16, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland (2016)
99. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-so-foreign Language for Data Processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. pp. 1099–1110. SIGMOD '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1376616.1376726

100. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: distributed reasoning over large-scale Semantic Web data. Web Semantics: Science, Services and Agents on the World Wide Web 7(4) (2009), http://www.websemanticsjournal.org/index.php/ps/article/view/173
101. Osorio, M., Aranda, C.B.: Storage Balancing in P2P Based Distributed RDF Data Stores. In: Proceedings of the Workshop on Decentralizing the Semantic Web 2017 co-located with 16th International Semantic Web Conference {(ISWC} 2017) (2017), http://ceur-ws.org/Vol-1934/contribution-04.pdf
102. Owens, A., Seaborne, A., Gibbins, N., schraefel, M.: Clustered TDB: A Clustered Triple Store for Jena (Nov 2008), http://eprints.soton.ac.uk/266974/, http://eprints.soton.ac.uk/266974/
103. Papailiou, N., Tsoumakos, D., Konstantinou, I., Karras, P., Koziris, N.: H2RDF+: An Efficient Data Management System for Big RDF Graphs. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. pp. 909–912. SIGMOD '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2588555.2594535
104. Peng, P., Zou, L., Chen, L., Zhao, D.: Query Workload-based RDF Graph Fragmentation and Allocation. In: Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016. pp. 377–388 (2016), https://doi.org/10.5441/002/edbt.2016.35
105. Peng, P., Zou, L., Özsu, M.T., Chen, L., Zhao, D.: Processing SPARQL Queries over Distributed RDF Graphs. The VLDB Journal 25(2), 243–268 (apr 2016), http://dx.doi.org/10.1007/s00778-015-0415-0
106. Penteado, R.R.M., Scroeder, R., Hara, C.S.: Exploring Controlled RDF Distribution. In: 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 160–167 (2016)
107. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Trans. Database Syst. 34(3), 16:1—-16:45 (Sep 2009), http://doi.acm.org/10.1145/1567274.1567278
108. Potter, A., Motik, B., Horrocks, I.: Querying Distributed RDF Graphs: The Effects of Partitioning. In: Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2014). pp. 29–44 (2014)
109. Potter, A., Motik, B., Nenov, Y., Horrocks, I.: Distributed RDF Query Answering with Dynamic Data Exchange. In: Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., Gil, Y. (eds.) The Semantic Web – ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I, pp. 480–497. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-46523-4_29
110. Prud'hommeaux, E., Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3c recommendation, W3C (2013), http://www.w3.org/TR/sparql11-query/
111. Przyjaciel-Zablocki, M., Schätzle, A., Lausen, G.: TriAL-QL: Distributed Processing of Navigational Queries. In: Proceedings of the 18th International Workshop on Web and Databases. pp. 48–54. WebDB'15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2767109.2767115
112. Przyjaciel-Zablocki, M., Schätzle, A., Lausen, G.: Querying Semantic Knowledge Bases with SQL-on-Hadoop. In: Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. pp. 4:1—-4:10. BeyondMR'17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3070607.3070610
113. Pujol, J.M., Erramilli, V., Rodriguez, P.: Divide and Conquer: Partitioning Online Social Networks. CoRR abs/0905.4 (2009), http://arxiv.org/abs/0905.4918

114. Punnoose, R., Crainiceanu, A., Rapp, D.: Rya: A scalable rdf triple store for the clouds. In: 1st Int. Workshop on Cloud Intelligence. pp. 4:1–4:8. ACM (2012)
115. Quilitz, B., Leser, U.: Querying Distributed RDF Data Sources with SPARQL. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) The Semantic Web: Research and Applications. pp. 524–538. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
116. Rohloff, K., Schantz, R.E.: High-performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-store. In: Programming Support Innovations for Emerging Distributed Applications. pp. 4:1—-4:5. PSI EtA '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1940747.1940751
117. Russell, J.: Getting Started with Impala: Interactive SQL for Apache Hadoop. O'Reilly Media (2014), https://books.google.de/books?id=o8SeBAAAQBAJ
118. Sakr, S., Wylot, M., Mutharaju, R., Le Phuoc, D., Fundulaki, I.: Linked Data: Storing, Querying, and Reasoning. Springer International Publishing, Cham, 1 edn. (2018), https://doi.org/10.1007/978-3-319-73515-3
119. Saleem, M., Mehmood, Q., Ngonga Ngomo, A.C.: FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In: Arenas, M., Corcho, O., Simperl, E., Strohmaier, M., D'Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thirunarayan, K., Thirunarayan, K., Staab, S. (eds.) The Semantic Web - ISWC 2015. pp. 52–69. Springer International Publishing, Cham (2015)
120. Saleem, M., Ngonga Ngomo, A.C., Xavier Parreira, J., Deus, H.F., Hauswirth, M.: DAW: Duplicate-AWare Federated Query Processing over the Web of Data. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) The Semantic Web – ISWC 2013: 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I, pp. 574–590. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-41335-3{_}36
121. Schätzle, A., Przyjaciel-Zablocki, M., Berberich, T., Lausen, G.: S2X: Graph-Parallel Querying of RDF with GraphX. In: Wang, F., Luo, G., Weng, C., Khan, A., Mitra, P., Yu, C. (eds.) Biomedical Data Management and Graph Online Querying. pp. 155–168. Springer International Publishing, Cham (2016)
122. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: Mapping SPARQL to Pig Latin. In: Proceedings of the International Workshop on Semantic Web Information Management. pp. 4:1—-4:8. SWIM '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1999299.1999303
123. Schätzle, A., Przyjaciel-Zablocki, M., Neu, A., Lausen, G.: Sempala: Interactive SPARQL Query Processing on Hadoop. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (eds.) The Semantic Web – ISWC 2014, Lecture Notes in Computer Science, vol. 8796, pp. 164–179. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-11964-9_11
124. Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: {S2RDF:} {RDF} Querying with {SPARQL} on Spark. PVLDB 9(10), 804–815 (2016), http://www.vldb.org/pvldb/vol9/p804-schaetzle.pdf
125. Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) The Semantic Web – ISWC 2011. pp. 585–600. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
126. Schmidt, M., Hornung, T., Meier, M., Pinkel, C., Lausen, G.: SP2Bench: A SPARQL Performance Benchmark. In: de Virgilio, R., Giunchiglia, F., Tanca, L. (eds.) Semantic Web Information Management: A Model-Based Perspective, pp. 371–393.

Springer Berlin Heidelberg, Berlin, Heidelberg (2010), https://doi.org/10.1007/978-3-642-04329-1{_}16
127. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization Techniques for Federated Query Processing on Linked Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) The Semantic Web – ISWC 2011. pp. 601–616. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
128. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–10 (2010)
129. Stein, R., Zacharias, V.: RDF on Cloud Number Nine. In: Ceri, S., Valle, E.D., Hendler, J., Huang, Z. (eds.) Proceedings of the 4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable {&} Dynamic. CEUR Workshop Proceedings (2010)
130. Stutz, P., Verman, M., Fischer, L., Bernstein, A.: TripleRush: a fast and scalable triple store. In: 9th International Workshop on Scalable Semantic Web Knowledge Base Systems. CEUR Workshop Proceedings, http://ceur-ws.org, Aachen, Germany (2013)
131. Stutz, P., Bernstein, A., Cohen, W.: Signal/Collect: Graph Algorithms for the (Semantic) Web. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) The Semantic Web – ISWC 2010. pp. 764–780. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
132. Stutz, P., Paudel, B., Verman, M., Bernstein, A.: Random Walk TripleRush: Asynchronous Graph Querying and Sampling. In: Proceedings of the 24th International Conference on World Wide Web. pp. 1034–1044. WWW '15, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland (2015), https://doi.org/10.1145/2736277.2741687
133. Wang, R., Chiu, K.: Optimizing Distributed RDF Triplestores via a Locally Indexed Graph Partitioning. In: Parallel Processing (ICPP), 2012 41st International Conference on. pp. 259–268 (Sep 2012)
134. Wang, X., Tiropanis, T., Davis, H.C.: LHD: Optimising Linked Data Query Processing Using Parallelisation. In: Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013 (2013), http://ceur-ws.org/Vol-996/papers/ldow2013-paper-06.pdf
135. White, T.: Hadoop: The Definitive Guide. O'Reilly, Beijing, 4 edn. (2015), https://www.safaribooksonline.com/library/view/hadoop-the-definitive/9781491901687/
136. Wilschut, A.N., Apers, P.M.G.: Dataflow query execution in a parallel main-memory environment. Distributed and Parallel Databases 1(1), 103–128 (jan 1993), https://doi.org/10.1007/BF01277522
137. Wu, B., Zhou, Y., Yuan, P., Liu, L., Jin, H.: Scalable SPARQL querying using path partitioning. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 795–806 (apr 2015)
138. Wu, B., Zhou, Y., Yuan, P., Jin, H., Liu, L.: SemStore: A Semantic-Preserving Distributed RDF Triple Store. In: CIKM-2014 (2014)
139. Wylot, M., Cudré-Mauroux, P.: Diplocloud: Efficient and scalable management of rdf data in the cloud. IEEE Transactions on Knowledge and Data Engineering 28(3), 659–674 (2016)
140. Wylot, M., Cudré-Mauroux, P.: DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud. IEEE Transactions on Knowledge and Data Engineering 28(3), 659–674 (2016)
141. Xu, Z., Chen, W., Gai, L., Wang, T.: SparkRDF: In-Memory Distributed RDF Management Framework for Large-Scale Social Data. In: Dong, X.L., Yu, X., Li, J., Sun, Y. (eds.)

Web-Age Information Management. pp. 337–349. Springer International Publishing, Cham (2015)
142. Yang, S., Yan, X., Zong, B., Khan, A.: Towards Effective Partition Management for Large Graphs. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. pp. 517–528. SIGMOD '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2213836.2213895
143. Yang, T., Chen, J., Wang, X., Chen, Y., Du, X.: Efficient SPARQL Query Evaluation via Automatic Data Partitioning. In: Meng, W., Feng, L., Bressan, S., Winiwarter, W., Song, W. (eds.) Database Systems for Advanced Applications. pp. 244–258. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
144. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster Computing with Working Sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing. p. 10. HotCloud'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1863103.1863113
145. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A Distributed Graph Engine for Web Scale RDF Data. PVLDB 6(4), 265–276 (Feb 2013)
146. Zhang, X., Chen, L., Tong, Y., Wang, M.: EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In: ICDE-2013. pp. 565–576 (Apr 2013)
147. Zhang, X., Chen, L., Wang, M.: Towards Efficient Join Processing over Large RDF Graph Using MapReduce. In: Ailamaki, A., Bowers, S. (eds.) Scientific and Statistical Database Management, Lecture Notes in Computer Science, vol. 7338, pp. 250–259. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-31235-9_16